Nicke – C Extensions for Programming on Distributed-Memory Machines

Dalia Malki
IBM Israel Scientific Center

Marc Snir
IBM Research, T. J. Watson Research Center

Abstract

This document describes the Nicke programming language. Nicke is an extension of C for programming distributed-memory machines. It supports both message-passing and shared-memory parallelism. We present the rationale for Nicke and describe the main features of the language. The Nicke language has been implemented on the 8CE machine under the Mach Operating System, [6], and is being ported to the Victor machine under the Trollius kernel [11, 5]. The language is supported by a preprocessor that translates Nicke code into standard C, and by a run-time library.

1 Introduction

1.1 Environment

The language Nicke is a parallel programming language that extends C. It is targeted for distributed memory multiprocessors, in particular, for the Victor prototype developed at IBM research [11]. It is equally well suited for other multicomputers, such as Ncube, iSPC2, etc. [2]. These are machines consisting of a large number of processors (256, for Victor), each executing its own independent code. Each processor has its own local memory, and communication between processors is done via message passing. Special hardware and firmware provide efficient support for message passing: In many of these machines, source to destination routing of messages is done transparently from the user, and the physical location of the communicating nodes has a relatively small effect on message transfer time. (This is not true for the Victor system, where message passing is handled in software) Still, interprocess communication is one or two orders of magnitude slower than access to local memory. Also, context switching is a relatively expensive operation: Fast microprocessors tend to have large “contexts” (register files, pipelines, on-chip caches, status registers, etc.) The explicit overhead of saving and restoring such information, and the implicit overhead of reduced cache performance, extract a large penalty from code with fine granularity.
In addition to this hardware environment, we also assume software that provides a few basic functions: Message transfer between physical nodes, memory allocation, and process spawning. These functions are supported by the Trollius kernel on Victor; similar functions are available on most commercial multicomputers.

The Nicke language supports the shared memory paradigm, in addition to the message passing paradigm. The Nicke language has been implemented to run under Mach, and is running on a shared memory multiprocessor, the 8CE machine [6]. However, most attention has been given in the design of Nicke to message passing support, and the current version of Nicke may not make the most efficient use of shared memory.

1.2 Rationale

The design of Nicke has been affected by many considerations, many of them mundane (such as the desire to implement the language using a simple preprocessor). However, the design broadly reflects our understanding of what is a reasonable high level language for the design and implementation of parallel algorithms. One obviously desires such language to be as high level as possible, abstracting as much as possible from the concrete details of the underlying machine. It is presumably easier to code correct programs in higher-level languages, because it is easier for the user to specify and understand the logical, qualitative effect of a computation. However, since performance matters to us, there is another iron-clad requirement: The high-level language should allow the user to easily specify and understand quantitative performance aspects of a computation, such as running time. Such is the case, for example, for sequential imperative languages such as Fortran or C: It is reasonably easy to estimate the number of instructions executed by a program; and computation time is approximately proportional to the number of high-level instructions executed. The frequent choice of Fortran or C for algorithm implementation (rather than languages with higher level operations or non-imperative languages) is due in large part to that transparency of the language. Similar considerations mandate that some features of parallel computations be explicit and easily quantifiable in Nicke.

The first such feature is parallelism itself, which we make explicit. There should be an obvious difference between code that is inherently sequential and code that can be executed in parallel, otherwise the programmer has scant support in designing parallel algorithms. A sequential language, with an underlying parallelizing compiler, will not do: The programmer would need a detailed understanding of the compiler in order to estimate performance.

The second such feature is communication. Communication is a severe bottleneck on the performance of large-scale multiprocessors. In order to achieve good performance it is important to optimize the computation to communication ratio. Such optimization sometimes require algorithm redesign, and can not be achieved by compiler alone. Thus, the programming language must distinguish between local communication (e.g., between a processor and its local memory), and global communication. In the shared memory paradigm, the local-to-global distinction is reflected by a distinction between private and shared variables. In the message passing paradigm this distinction is reflected by using message passing operations for interprocess communication. Nicke supports both paradigms, and makes global communication explicit.
The third feature is synchronization. Large MIMD machines are not synchronous, and synchronization is an explicit, and often expensive operation. This is especially true if arbitrary subsets of processors are to synchronize.

Theoretical and pragmatical considerations indicate that other, more detailed features of a parallel computation can, ideally, be hidden from the user, and handled by compiler, run-time resource manager, or firmware, with a predictable and acceptable effect on performance. For example, it is feasible to ignore the precise number of physical processors, and program for "virtual processors" (processes). This holds true when the program has slack, i.e. the amount of available parallelism is larger than the number of physical processors, the overhead of time multiplexing each processor is acceptable, and sufficient memory is available at each node. (The time-multiplexing of several virtual processors on one physical processor may be programmed at compile time, with very small overhead [7]; it may even be supported by hardware [9].) It is possible to generate dynamically processes, and allocate them to physical processors, at a cost similar to the cost of a global synchronization. It is feasible to ignore the detailed topology of the interconnection mechanism, and merely distinguish "local" from "global". This assumes again that the program has slack, and that communication latency can be hidden by time-multiplexing each processor.

Thus, in an ideal world, a programming language with explicit parallelism and dynamic process creation, global vs local distinction, and explicit synchronization operations, would be sufficient for the design and implementation of efficient parallel algorithms. However, we do not live in an ideal world, and current system support for parallelism leaves much to be desired. Moreover, even in an ideal world it is often necessary to do low-level tuning, and exercise more explicit control on physical resources. We desire to do such low-level tuning within the same programming environment that is used for "high-level" programming. However, we wish to keep the distinction between the high-level, machine-independent, virtual model used for algorithm design, and the low-level, machine-dependent, physical model used for performance tuning.

To support this point of view, Nicke is designed to consist of two layers: The virtual layer and the physical layer. At the virtual layer a computation is expressed in terms of processes that can be spawned dynamically. These processes communicate via messages or shared variables. At the physical layer, the user may control which physical processors are allocated to each process, and may control the policy that is used to emulate shared memory. A user may totally ignore the physical layer, and code his or her application entirely in the virtual layer; some default resource allocation mechanisms will be used. It may then improve the performance of the code by exercising more control on resource allocation. This, ideally, will not change the outcome of the computation, but will just reduce its cost. (In practice, a different outcome may occur if the code is nondeterministic, or if an exception occurs because the program exhausts some resource.)

This two-layered approach is not new. This is the essence of metaprogramming in logic or functional programming [8, 3]. Similar goals are achieved by pragmas, or compiler directives in many parallel programming languages. Our system is somewhat more flexible in that its pragmas are executable at run time.
2 Basic Constructs

2.1 Processes

2.1.1 Process declaration

The basic unit of execution in Nicke is the process. Nicke processes are defined as C functions with the additional keyword procdef, designating them as processes. Process functions may not have parameters. Here is an example of a process declaration (the body omitted):

```c
procdef F()
{
    ...
}
```

2.1.2 Process creation

Processes are created in two ways: by process declaration and by dynamic process generation.

A process declaration creates a process group, and assigns a name to each process in the group. The processes are instantiated when the declaration is elaborated: global processes are created at startup and automatic processes within functions are created upon function invocation. For example, the following declaration instantiates a group of 100 processes of the above type F():

```c
procreate <F f[100]>
```

The procreate declaration may also be used to create non-homogeneous groups of processes, as in the following tuple declaration:

```c
procreate <F f1, G g1, H h1>
```

Process generation statements are used for dynamic creation of processes. They appear in the code and are evaluated at runtime. Following are examples of the two forms of calls:

```c
parc(F[100])
parw(F, G, H)
```

In the parc version the parent (calling) process continues executing with the created processes, while in the parw version it waits for their termination. The runtime call permits any runtime computable expression to be used as array dimension, allowing calls of the form:

```c
parw(F[n])
```

The local constant INDEX is initialized to the index of the process within its group when the process is created. Thus, siblings can immediately proceed to perform distinct computing tasks, according to the value of their index, without further coordination. This is in contradistinction to the situation in Ada [12].
2.1.3 Remote references

A reference to a process is, in effect, a network-wide logical address that can be used to communicate with this process; each process that has a copy of that reference can communicate with the referenced process. A process declaration associates a name with each declared process. This variable refers to the process. Code within the scope of this declaration can use this reference to communicate with the referenced process. In a process generation statement, no names are assigned with the created processes; however the statement returns a reference (pointer) to the process group. The function member(group-ptr, i) returns a pointer to the \( i \)th process in the group referenced by group-ptr. Thus, the parent of a process group can communicate with each process it spawns. The constants GROUP, ME and PARENT are initialized at process creation to refer to the process group, the process itself, and the process parent, respectively. Thus, a process can communicate with its parent, and with any of its siblings.

References to processes and process groups have type remptr (remote pointer). A Nicke program can declare variables of this type and can assign them values; references to processes or process groups can be communicated by passing the value of such remote pointer.

2.2 Interprocess Communication

2.2.1 Scoping rules and sharing

The regular scoping rules of C apply to Nicke. In particular, a process can refer to any externally declared variable within its scope. However, the basic paradigm of Nicke is that of distributed memory: each process has its own address space. Externally declared variables are not truly shared: each process may have its own copy of such variables. If an external variable is initialized when declared, and is not modified anywhere during execution, then any access to this variable will return the correct, initial value. The effect of updates to external variables is undefined. Thus, external variables can not be used for interprocess communication, but only as system-wide constants.

2.2.2 Synchronization

Nicke provides several operations to synchronize siblings within a group. The sync instruction implements a barrier synchronization: each sibling reaching a sync suspends until all siblings have reached a sync, or have terminated. a pbreak instruction allows to terminate execution of all processes in a group.

2.2.3 Messages

The basic communication mechanism between processes is asynchronous message passing. Messages are addressed to processes via process pointers. The sending command is nonblocking. The parameters of a send() are the addressee, a message tag and a variable list. The variable list contains any legal expression passable as a parameter to function. The message also carries the sender address.
There is a blocking receive() command, where the receiving process suspends until
a suitable message is available, and a non-blocking receivec() command, where a zero
value is returned if no suitable message is available. The parameters are a receive condition
and a list of reception variables. The list of reception variables may include any legal l-
value. The sending list and the receiving list must match in length and types (the results
in the case of nonmatching lists are undefined). The receive condition is an arbitrary
expression that may involve the constants TAG (the message tag) and SENDER (the message
sender). (The current implementation supports only a restricted form of conditions.)

Various communication mechanisms (synchronous message passing, remote procedure
call) can be implemented on top of this simple message-passing mechanisms. It is expected
that these will be supported by library extensions.

2.3 Example

The constructs introduced in this section are sufficient to program nontrivial parallel
applications. The following is an example of the obvious (and inefficient) parallel version
of the sieve of Erasthotenes, coded in Nicke. A sieve process tests successive numbers
for division by a prime p. The sieve processes are linked in a chain, and each number is
passed along this chain. A number p that survives all tests is a prime. A new process
is appended to the end of the chain to test for division by p, and p is sent to the print
server process. The program has no termination condition, and will end when running
out of resources.

```nicke
procdef PRINT() {

    ... 
}

procreate <PRINT print>;  /* create print server */

procdef SIEVE() {
    int i, p;
    remptr next;

    next = NULL;
    receive(i, p);

    while(1) {
        receive(i, i);  /* receive new candidate */
        if (((i%p) != 0) { /* local test succeeds */
            if (next == NULL) { /* new prime found */
                send(print, 1, i);  /* send new prime to print process */
                next = group(parc(SIEVE), 0);  /* create next process */
```
} procreate <SIEVE head>; /* create first sieve process */

main()
{
  int j;

  send(print,1,2);
  for(j=2; j++;)
    send(head,1,j); /* send numbers thru sieve */
}

2.4 Processor Allocation

A Nicke program can specify which processor is to run a process by adding a place(exp) statement to the process generating declaration or instruction. This expression may involve the pseudo-constant INDEX. The placement expression is evaluated once for each generated process in the process group, with INDEX set to the process index. The resulting value is the index of the processor that is allocated to the process. If place() occurs in a process declaration, then the place instruction should be a constant expression; if it occurs in a process generation statement, then it can be an arbitrary expression.

For example, assume that the system has NUMPROC processors, numbered from 0 to NUMPROC-1, and that processor i is close to processors i-1 and i+1. Then it would be advantageous, in the previous example, to allocate successive sieve processes at successive processors (with wraparound). The statement that spawns new sieve processes is modified accordingly:

    next = parc(SIEVE) place((Location(ME) + 1)%NUMPROC)

Alternatively, we might want to use random placement, to achieve better load balancing. This would be achieved by a place statement of the form

    next = parc(SIEVE) place(random()%NUMPROC)

3 Shared Variables

3.1 Rationale

Shared variables are a very natural idiom for the expression of many parallel algorithms. Think about the simulation of a large physical system (or, equivalently, of the evolution of a PDE solver). A large, shared data structure represents the state of this system
at a particular time; a (parallel) computation pushes simulated time ahead (possibly, at different rates at different places). One thinks of such algorithms as performing a coordinated computation on a representation of the data domain of the problem. The possible partition of this representation into pieces allocated to distinct processors, so as to decrease communication, is accessory. The first, natural encoding of the algorithm would use a shared data structure to represent the global data domain. A language that does not support this idiom seriously handicaps its user.

However, efficient partition of data is essential to performance on a multicomputer where inter-process communication is orders of magnitudes slower then local memory access. Thus, we would like to preserve the paradigm of shared data structures, in particular shared arrays, while providing means to optimize access to such shared structures. The goal is to provide mechanisms to support shared data structures on a multicomputer such that:

- Access to shared variables is always “correct” (i.e., serializable).
- Access to shared variables is efficient when the algorithm exhibits much locality.

Caching techniques are very efficient in exploiting temporal locality of accesses; however, by themselves, they do no lead to a good exploitation of spatial locality. The long latencies in message-passing multicomputers imply that data must be transferred in large chunks (hundreds or thousands of bytes), to amortize the fixed overheads. Such chunking can be often achieved by partitioning the data correctly. However, the basic chunks will often not reside in contiguous segments (for example, they may be submatrices of a two dimensional array). In such case, hardware blocking techniques are inefficient. For many interesting numerical algorithms computation is not evenly spread over all the data domain: adaptive algorithms spend more computing time in “interesting” parts of the domain; sparse arrays may be used to delete from the data representation “uninteresting” parts of the data domain. The correct domain partition, in such cases, is irregular and data dependent. Such partitions do not have a simple syntactic description, nor can such partition be derived by compile time analysis. Thus, we want to allow the user more flexibility in specifying partitions of shared data structures in an algorithm dependent manner. For a similarly motivated research, see [10].

3.2 Shared arrays

Nicke currently support one type of shared data structure: shared arrays. Shared arrays are declared in the same way as regular C arrays, except that the declaration is prefixed by the keyword shared, must be external and can not be initialized. Thus

```
shared int a[100][100]
```

creates a two dimensional, 100 × 100 shared array of integers. The elements of the array are regular C variables (of type integer, in this example). However, the shared array has a very different structure from a regular C array. In particular, one can not use pointers to access elements of a shared array, as usually done in C – entries are always accessed by their indices.

An array entry can be assigned a value, by a regular assignment, e.g.
a[i][j] = 5

or by an assignment operator, e.g.

a[i][j] += 3

This increments the value of a[i][j] by 3. An array entry can be used like any other C variable of the corresponding type. (To simplify preprocessing, the current version of Nicke uses distinct separators for operations on shared variables.)

Assignments are executed atomically. Consider, for example the following code.

shared int a;

procd SILLY()
{
    printf("%d", a);
    a += 1;
}

main()
{
    a = 0;
    parw(SILLY[3]); /* spawn three processes that each increments a by 1 */
}

The variable a is thrice incremented atomically by one. It successively assumes the values 0, 1, 2 and 3. This code will print the three values 000, 001, 011, or 012, in some permuted order.

3.3 Shared array partition

The Nicke language provides several tools to help the user specify a partition of a shared array, and specify a protocol for shared data access.

3.3.1 Windows

A shared array can be partitioned into (typically disjoint) windows. A window can be thought as of a cache line, storing part of the array. Remote accesses that involve data transfer will usually move the entire window data. The partition of a shared array into windows is specified when the array is declared by appending a window type to this declaration. Thus, the declaration

shared int a[100][100] subarray(2,5)

specifies that the array a[100][100] is partitioned into windows each consisting of a 2 × 5 submatrix. Syntactically, the window type is a function which includes constant parameters. Nicke has a library of predefined window types, corresponding to the most
usual array partitions. The (expert) user may add new window types to this library. For each added window type, the user has to supply a set of routines for the initialization of the window structure, and for the access of elements. Basically, array entries are stored in a “window major” order; an entry is accessed by its window index, and its displacement within the window. The window access routines compute the window index and displacement from the array indices. Typically, these will be simple, inlined routines, so that the overhead per entry access is small, when the entry is available locally. However, the user may specify more complex partitions; in particular there are no impediments to the use dynamically changing partitions, or overlapping partitions, other that the potentially large overhead of window routines for such partitions.

3.3.2 Caching protocols

Each window in a partitioned shared array is managed by a fixed processor, the window manager. The user may specify the location of the window manager for each window, by adding a placement expression to the shared array declaration; this expression has the same syntax and semantics as the placement expression for processes. One simple use of partition and placement is to statically allocate each window to a fixed processor in the network. An access to a shared variable within a window will involve a few instructions, if the window is at the node where the access is done; it will require a message exchange with the window manager, otherwise. Nicke also supports various caching policies for windows. One can have a single copy caching protocol (which we call erew): whenever a processor accesses a shared variable then the (unique) copy of the window containing this variable is moved to the accessing processor. Typically, this requires a protocol where the requester communicates with the window manager, and next with the processor currently holding the window. Another supported protocol (crew) allows multiple copies per window: whenever a processor accesses a shared variable, it receives a copy of the window containing the variable; if it is a write access, then all other copies are invalidated. Other caching protocols (including “non consistency check”, called chaos) are also supported. The type of protocol to be used may be specified when the shared array is declared.

3.3.3 Explicit window manipulation

Clearly, indiscriminate use of window caching protocols may result in high overheads. It is expected that tolerably low overheads will be obtained by tailoring the data partition and the caching protocol to the communication pattern of the algorithm. As a last resort, Nicke also allows explicit manipulation of windows by the user, with instructions such as fetch, flush and invalidate.

4 Implementation

The implementation of Nicke consists of two parts:

- The preprocessor that translates a Nicke program into standard C code, containing additional calls to system and library functions.
• The run-time support, which consists of a set of functions providing various communication and resource allocation functions.

The Nicke language supports global object spaces for processes, process groups and shared arrays. Nicke processes may require services from any network location using remote reference pointers. A set of data structures link between the program level and the runtime support: they are recognized by the preprocessor and manipulated by the runtime servers. The servers are resident at each node in the network and provide network-wide services.

4.1 Preprocessor
4.1.1 Data Structures

The basic structure used to access any remote object is the remote pointer structure, called remptr. The referred object type is determined by the mode field. The structure is of the following type:

```c
typedef struct {
    t_location location;
    union {
        int    pid;
        int    gid;
        int    tag;
        char   *addr;
    } id;
    int    mode;
} remptr;
```

A process environment structure accompanies each Nicke process. It holds Nicke-specific context information for the process. The preprocessor passes down this structure through every function call as an additional parameter, and makes the following information available for the process:

```c
typedef struct {
    remptr    e_parent;
    remptr    e_me;
    remptr    e_group;
    int       e_groupsize;
    int       e_groupind;
} processenv;
```

References to all the constants INDEX, ME, GROUP, PARENT, etc., are replaced by the preprocessor with the appropriate fields in the environment structure.
4.1.2 Parallel Constructs

The preprocessor translates most Nicke constructs into library routine calls that implement the services directly.

Processes

Declarations that create processes are translated into code that creates the process group and spawns these processes. A process array declaration is translated into a for-loop; a process tuple declaration is translated into a sequence of statements. Externally declared processes are spawned by the main process that executes on processor HOST.

Nicke processes are always created within process groups. The spawning of a child proceeds as follows:

- The user code allocates a new process group of the appropriate size and receives back a remote pointer for the group.

- For each process in the group, the N.RemSpawn() routine is called. If allocation and/or placement occur in the declaration, they are computed here and passed as parameters to N.RemSpawn. The routine prepares an environment structure for the new process containing the group pointer, the parent (calling) process pointer, the group size and the process' index in the group. It calls the remote kernel which determines the local process id and adds it to the environment before invoking the actual process body function. The remote kernel returns the process id in a message. N.RemSpawn() finally returns the caller a remote pointer to the newly created process.

- The kernel at the child node spawns the child process.

- The remote pointer returned by N.RemSpawn() is assigned to the process array variable and the process is added to its group via the N.AddMember() call.

Process Groups

All the group operations are implemented via single library routine calls:

- The barrier synchronization sync is handled by calling N.Synch() from all the processes in the group.

- Similarly, pbreak is handled by N.Pbreak().

- The parw call incurs an implicit call to N.WaitGroup() which blocks until all the children have terminated.

- A new process group is created by first allocating it via the N.NewGroup() call and then adding the members one by one via N.AddMember().

Messages

A message send operation encapsulates the data in a buffer and calls the sending routine; a complement receive operation moves the data into the variables in the receive list.

Shared Arrays

The declaration of a shared memory array is translated in two parts:
• A declaration of a global window descriptor structure of the appropriate type, declared in all the nodes.

• The initialization of the window descriptor structure and allocation of the actual memory portions for it. The initialization routine is invoked concurrently on all nodes, each allocating the local portions and assigning global pointers accordingly.

Accessing a shared memory element may be either for read or for write. In both cases, the preprocessor calls the \texttt{index} and \texttt{offset} macros of the given window type to attain a window handle and an offset within it. It invokes either \texttt{N.ShGet()} or \texttt{N.ShPut()} with a buffer address and a length parameter to perform the copy in/out.

\subsection{Compile Time Tags}
Nicke allows declaration of processes and shared arrays in the global program scope. The variables associated with these objects should be valid in all the network nodes. These variables are remote pointers, and their values are bound at runtime, after the appropriate resource allocation occurs (and at the location of allocation). Nicke uses remote tags as the valid values for pointers. The tags are determined at compile time, and are used locally for tagging objects. Since the location is known at compile time, the pointers are valid everywhere once the resource is allocated and tagged.

\section{Nicke kernel}
The runtime library and resident kernel support the compound operations required by the preprocessor.

The underlying kernel layer is assumed to support certain, basic intra-node primitives. If the target system contains a compatible version of these services, they may be used. Otherwise, they need to be implemented for the Nicke library to run with. The required services consist of a local process spawner that maintains information about the living processes and provides the process control operations \texttt{yield, terminate, sleep, wakeup} (in any form). It includes a memory manager that manages the entire local memory space. And it includes a router that forwards messages to any node in the network. (This layer is given as a standard part of the Trollius system; the Trollius version of Nicke uses Trollius processes for each Nicke process.)

On top of this layer, the Nicke kernel maintains its own notion of processes, provides a memory tagging facility, and provides network wide services through memory resident servers.

The spawning of remote processes is carried out by the \texttt{spawner server}. The server prepares the Nicke context for new processes and supports running of program parts as separate processes.

The \texttt{group server} maintains group information at the parent location. It provides the group operations like member queries and barrier synchronization.

The \texttt{window server} manipulates memory windows and provides software caching operations. These include caching, replication, locking and overlap handling. All shared array operations are carried out through the window interface and allow the system to modify
the window information internally (while the user only maintains a window handle for reference). In this way, the system may operate with a limited amount of memory, swap window contents to disk and perform periodic garbage collection.

The Nicke library interface routines invoke the operations supported by the servers. The library interface routines perform the encapsulation of parameters needed for the service requests into messages and the message exchange. For instance, whenever a response is required from the server the appropriate interface routine automatically attaches the "return-address" to the request.

5 Conclusion

The runtime support package for Nicke has been implemented on 8CE, an experimental shared bus multiprocessor workstation built at the IBM T. J. Watson Research Center [6]. The Mach [1] operating system runs on 8CE and provides multithreads support and shared memory handling at the kernel level. The Nicke runtime package uses the Mach C Threads facility for multiprocessing [4].

The preprocessor can run on any Unix machine, and produces standard C code with library routine calls. The current implementation of the preprocessor poses some restrictions on the Nicke syntax.

The Victor V256 system has been the first target, message passing machine for Nicke [11]. Victor is a 256 Tranputer machine, interconnected in a mesh topology and connected to an IBM RT serving as host. The Trollius system provides basic operating system services on Victor [5]. The Trollius kernel is loaded from the host onto every node in the network. The resident kernel supports asynchronous message passing and routing, process spawning and memory handling. The Nicke runtime support package has been implemented on top of Trollius, using a Trollius process for each Nicke process and using the Trollius message passing system for interaction.

As can be inferred from this paper, our goal is not the design of the most clean, elegant, high-level language for parallelism (an extension of C can not possibly be clean and elegant). Rather, we try to create a flexible tool for experimenting with algorithmic aspects of parallel computation.

A language with better typing and better extensibility than C would provide a better basis to our work. At present, constraints of the C language prevents us from making process types, processes and shared variables first-class citizens in Nicke. The use of C++ would improve, but not completely solve this problem. Also, several compromises were made in the current prototype to avoid the need for a full Nicke compiler (the preprocessor does not create a symbol table). While not affecting the semantics of the language, these make the syntax bulkier, and introduce annoying restrictions.

The Nicke kernel has not been optimized for performance, and we expect much work to be involved in kernel tuning. The kernel is parallel in that servers may run concurrently at each processor; however, each kernel call is executed sequentially, by one server. Further parallelism could by applied to distribute execution of kernel calls. Also, the design of efficient window management algorithms for shared arrays is a challenging research issue.

We have not given so far any attention to I/O. Standard Unix I/O services will not be
appropriate for a large scale parallel machine with massively parallel I/O. New paradigms
for parallel I/O need to be developed and reflected at the programming language level (or
the metaprogramming level).

Finally, the two-layered approach exemplified by Nicke may offer a useful direction for
the (semi) standardization of parallel programming languages, e.g. for the definition of a
standard parallel C language. At present, parallel architectures are too different to hope
that a unique parallel language will map equally well say, on a Victor machine or an IBM
3090. However, this does not preclude the use of a common, core language that would be
appropriate for a wide variety of parallel machines. This core corresponds to the virtual
layer of Nicke. In order to achieve good performance it would be nessesary to annotate
the code, providing more machine-specific information. The annotation language may
differ from one type of architecture to another (but not necessarily from one machine to
another, when the machines are broadly similar). This annotation mechanism corresponds
to the physical layer of Nicke. With such approach, code can be ported from one machine
to another by mere recompilation. The code can then be tuned for improved performance,
without affecting the computation outcome.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach:
University, August 1986.


environment multiprocessor workstation. Technical Report TR 14491, IBM Research,


programming environment for parallel and distributed computing. Technical Report
YALEU/DCS/RR-524, Yale University, dept. of Computer Science, 1987.

