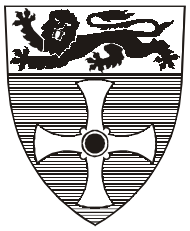
UNIVERSITY OF
NEWCASTLE



COMPUTING SCIENCE

NIP: A Parallel Object-Oriented Computational Model

Paul Watson and Savas Parastatidis

TECHNICAL REPORT SERIES

No. CS-TR-658
November, 1998

Contact:
Paul.Watson@ncl.ac.uk, Savas.Parastatidis@ncl.ac.uk
http://www.cs.ncl.ac.uk/people/paul.watson
http://www.students.ncl.ac.uk/savas.parastatidis

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

NIP: A Parallel Object-Oriented Computational Model

Paul Watson and Savas Parastatidis
{Paul.Watson,Savas.Parastatidis}@newcastle.ac.uk
Department of Computing Science,
University of Newcastle-upon-Tyne, Newcastle NE1 7RU, UK

Abstract

Implicitly parallel programming languages place the burden of exploiting and managing parallelism upon the compiler and runtime system, rather than on the programmer. This paper describes the design of NIP, a runtime system for supporting implicit parallelism in languages which combine both functional and object-oriented programming. NIP is designed for scaleable distributed memory systems including networks of workstations and custom parallel machines. The key components of NIP are: a parallel task execution unit which includes a novel and efficient method for lazily creating parallel tasks from loop iterations; a novel distributed shared memory system optimised for parallel object-oriented programs; and a load balancing system for distributing work over the nodes of the parallel system. The paper describes the requirements placed on the runtime system by an implicitly parallel language and then details the design of the components that comprise NIP, showing how the components meet these requirements. Performance results for NIP running programs on a network of workstations are presented and analysed.

Keywords: Implicit Parallelism; Networks of Workstations; Lazy Task Creation; Distributed Shared Memory.

1. INTRODUCTION

This paper describes work aimed at reducing the effort required to develop efficient programs for parallel architectures, in particular those with scaleable distributed memory architectures. Many practitioners believe that the widely used message passing methodology, currently embodied by PVM [1] and MPI [2], is too complex and time-consuming [3]. In order to write a parallel program using these libraries it is the responsibility of the programmer to:

- Divide the problem into parallel tasks and spread them across the nodes of the parallel system, to attempt to balance out the computation as evenly as possible.
- Ensure that the data moves with the computation. PVM and MPI, for example, do not support shared inter-node memory address spaces; thus, data must be explicitly sent in messages from tasks that produce data to those which consume it.
- Deal with any consistency problems caused when multiple tasks simultaneously try to alter shared data.
- Synchronise tasks, where necessary, by passing messages.

Programmers, therefore, have the responsibility for making all the important decisions about how to efficiently exploit parallelism. This has major disadvantages:

- Writing parallel programs requires specialist expertise (parallel programmers are often seen as the "rocket scientists" of the programming world), it involves excessive development effort and it is error prone.

- For some important types of computation, usually dynamic in nature, it is very difficult to evenly balance the load across all the processors. The degree of parallelism and the time taken to execute a task may be data dependent, making it impossible for a programmer to, statically or dynamically, work out how to divide the problem evenly across the nodes.
- In theory, programs written using PVM or MPI are portable and will run on any system, which supports PVM or MPI. In practise, however, the programmer has to make choices, often for tuning purposes, which result in the program performing better on some platforms than others. A key example is that the programmer must choose a task granularity which is appropriate for the target platform.

These problems are becoming more generally recognised. For example, investigators examining the design of future Petaflops systems [3], when discussing message passing systems such as MPI and PVM, state that "Unfortunately, developing large-scale parallel programs using these packages is very difficult and time-consuming." They also predict that a parallelism of up to 1 million will be required to achieve Petaflops performance. A programmer cannot be expected to explicitly create and manage this degree of parallelism except for the most regular of computations.

For these reasons, the design of an implicit parallel system has been investigated in which it is the compiler and runtime system, rather than the programmer, that are responsible for the creation and management of parallelism. The runtime system, called NIP, has been designed and a prototype of it has been implemented. NIP removes from the programmer the burdens of dividing a program into parallel tasks, sharing work evenly across all the nodes, organising communication between tasks, making the data available to tasks that require it, and synchronising tasks. This paper describes the design of the NIP system, its prototype implementation, and gives a preliminary performance evaluation.

The rest of the paper is organised as follows: Section 2 discusses high-level programming languages which support implicit parallelism. The attributes of the distributed memory parallel architectures that are targeted are described in section 3. The requirements placed on NIP and an overview of its structure are introduced in section 4. In section 5, the NIP execution model and the techniques that incorporates are described in more detail: a novel method for cheaply, lazily extracting parallelism from loops, the task execution mechanism, and dynamic load balancing. Then, the Distributed Shared Memory is presented (section 6) which includes a novel method for optimising performance by combining object locking with cache coherency. Preliminary performance results for the NIP prototype implementation are described in section 6.1. Finally, conclusions from the work are drawn in section 8.

2. PARALLEL PROGRAMMING LANGUAGES

In the opinion of many software developers, functional programming languages have a number of advantages over conventional, imperative languages including their expressiveness, and their amenity to reasoning about semantics. However, they also have further advantages for parallel computation. In particular, functional programs contain far fewer constraints on execution order than do their imperative counterparts. This is because all expressions in a functional program are referentially transparent [4]. Therefore, the order of their execution cannot affect the result of the execution, and this increases the scope for parallel execution. Consequently, a number of parallel functional programming systems have been developed over the last 20 years [5-7] to attempt to exploit these advantages.

Unfortunately, the very property that makes functional programs so well matched to parallel systems—referential transparency—makes the programming of certain important classes of computations unnatural, contorted and complex. In particular, many computations (or parts thereof) are naturally expressed through an object-oriented programming style in which objects encapsulate state which may be updated through method calls. Method calls to objects may not be referentially transparent, as identical calls can return different values, and therefore these types of computations cannot be directly expressed in a functional program.

As it was felt that functional programming was ideal for many types of computations, and well suited to parallelism, it was not rejected entirely as the preferred programming methodology for NIP over object-orientation. The UFO (United Functions and Objects) [8] programming language was adopted as it brings together the functional and object programming styles in an elegant way. In UFO, a clear distinction is made between objects containing mutable state and objects containing state that does not change after construction. The latter can be used to write purely functional programs but the two types of objects can also be freely combined within a program. In order to gain the benefits of parallel functional programming described above, as much as possible of the computation should be expressed through purely functional objects. However, where it is natural to do so, mutable objects can be constructed and manipulated.

The NIP runtime system was designed and implemented to support the execution of programming languages like UFO.[1] However, the mechanisms and solutions in NIP could have application across a wide range of parallel languages, especially those based on functional or object-oriented programming.

The rest of this section highlights those attributes of UFO that influenced in the design of NIP, while the rest of the paper concentrates on the NIP runtime system.

2.1. Objects in UFO

In UFO, the class interface may contain *functions* and/or *procedures*. A function call on an object cannot alter the state of the object, but a procedure call can. Instances of a class with only function calls are regarded as *immutable objects* because their state cannot be changed once they are constructed. In contrast, instances of a class with at least one procedure call are described as *mutable objects*.

The semantics of UFO state that the execution of a procedure call on an object should prevent other procedure or function calls from being executed on the same object at the same time. However, function calls, though, on the same object may be executed in parallel.

2.2. Opportunities for Exploiting Parallelism in UFO

As a language that supports implicit parallelism, UFO does not provide the means for a programmer to explicitly create parallel tasks. Instead, the syntax and semantics of the language provide two opportunities for the compiler and runtime system to create and exploit parallel tasks:

- In UFO, all calls (functions and procedures) are strict on all their arguments—the arguments must be first evaluated before the call can proceed. If there are two or more arguments, the option of evaluating them in parallel is presented. Once the execution of all arguments has been completed, then the body of the call can be executed. It is

[1] A full description of the UFO language is outside the scope of this paper. The interested reader is referred to [8].

important to note that as UFO is an implicit parallel language the decision to execute the arguments in parallel or serially will be made by the compiler and/or the runtime system rather than the programmer.

- Where there is a loop, tasks can be created to execute independent iterations in parallel. Figure 1 shows the definition of a loop in UFO. The map function takes an array and a function as its arguments and applies the function to each element of the array. Each iteration is independent and so could be executed in parallel.

```
map(a: Array[T];f:T->U): Array[U] is
   for x in a do
      y = f(x) return y
   od
```

Figure 1: A loop in UFO.

3. PARALLEL MACHINE ARCHITECTURES

While NIP is intended to be portable across different classes of parallel architectures, it primarily targets the distributed memory multiprocessor architecture (Figure 2). The architecture consists of a set of uniprocessors or multiprocessors connected by a high performance network. The main constraint of the architecture is that each processor can only access its local memory, and so all communication between nodes is via explicit messages sent over the network.

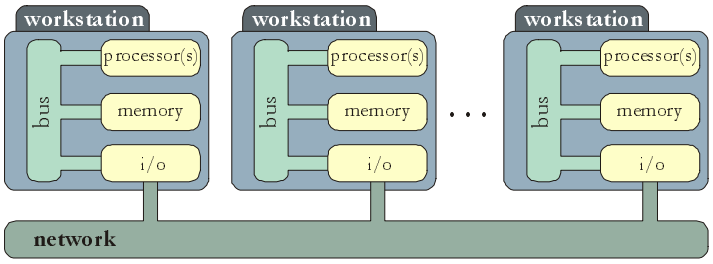


Figure 2: The Distributed Memory Architecture.

Distributed memory multiprocessors are high scalable; when a node is added to the system, all the key performance attributes are scaled, including processing, bus, memory, and IO (throughput and capacity). However, the distributed memory architecture presents more difficulties to the parallel application programmer than both the shared memory multiprocessor and the uniprocessor. The programmer has to subdivide the program into units that run on different nodes and communicate via messages sent over the network.

Whilst NIP does not target any particular distributed memory parallel system the system used for the development and benchmarking of NIP is the APP (Affordable Parallel Platform), which is built entirely from standard, low-cost, commodity hardware components.

Current commercially available MPP (Massively Parallel Processing) systems are also targets for NIP. However, the custom designed components of the MPP systems (including internal networks, processor-network interfaces and cabinetry) lead to high design, development, and manufacturing costs which are passed on to the user. Consequently, they have a cost-performance ratio, which is significantly higher than that of commodity uniprocessor systems. In addition, their high absolute cost restricts the number of users who can afford them.

The APP addresses the problem of cost by using standard, low-cost, commodity hardware components; specifically high performance PCs interconnected by a high throughput,

scaleable ATM network (for experimental reasons, the PCs are also interconnected by Fast Ethernet). This approach could allow high performance systems to be built at a significantly lower cost than current parallel systems.

4. NIP DESIGN OVERVIEW

The goal of NIP is to execute compiled UFO programs exploiting parallelism as efficiently as possible. In this section, an architectural overview of NIP is given. It begins with a description of NIP running on an abstract parallel system and then describes how it maps onto a distributed memory parallel system such as a network of workstations.

4.1. The NIP Abstract Parallel System

The NIP runtime system can be considered as an abstract parallel system consisting of a *load balancer* and an *object memory*. The unit of execution is a *task* and there is a limit to the number of tasks that can be executing in parallel at a particular time. When the limit is reached, the system is considered "busy" as the parallelism it offers is fully exploited. The load balancer is responsible for keeping the system at the "busy" state by creating new tasks when it is necessary as is now described.

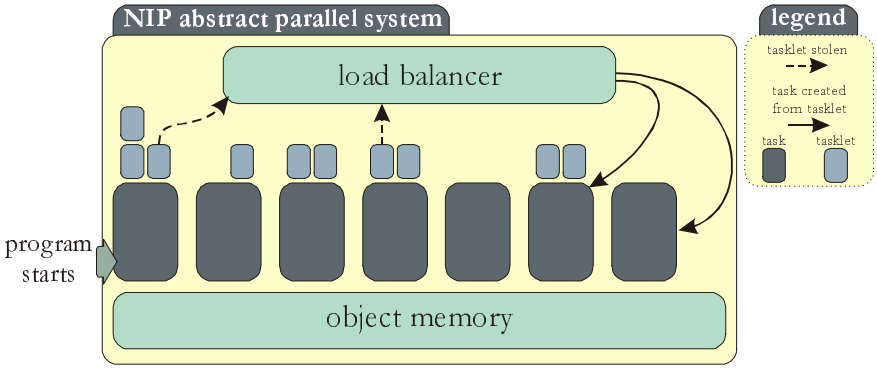


Figure 3: The NIP abstract parallel system.

The UFO programmer can develop an application for the NIP abstract parallel system without being concerned about how to exploit parallelism. The UFO compiler will identify those parts of the computation that can be evaluated in parallel. However, it will not create new tasks for them; instead, it will plant code to create *tasklets*. Tasklets are small representations of pieces of computation that could be evaluated in parallel. Computationally, tasklets are cheaper to create than tasks. When work is required to keep the system busy, the load balancer can use the information encapsulated in a tasklet in order to create a new task.

NIP will automatically create a single task when the execution of the UFO program starts. If the program is parallel, the task will produce tasklets which the load balancer converts into tasks until the system is "busy." The newly created tasks may also themselves produce tasklets that are available to the load balancer. If a tasklet has not been converted to a parallel task by the time the result of its computation is required, the task that created the tasklet removes it and executes the computation serially. The technique used by the load balancer to dynamically create tasks only when necessary is called *lazy task creation* [9] and it is discussed in detail in section 5.

4.2. Task States

A task may be in any of the four states shown in Figure 4. When a task is created, it always starts in the "running" state. However, there may be situations when the task will have to become "suspended" (section 5). In such a case, if necessary, the load balancer will try to create a new task in order to keep the system as close to the "busy" state as possible. If a task can exit the "suspended" state and the parallel system is not "busy," the load balancer will return the task in the "running" state. Otherwise, the task will enter the "waiting" state. When the load balancer requires a task to run, it first checks whether there are any tasks in the "waiting" state, before creating a new one from a tasklet, and if so it moves the task to the "running" state. Finally, a task that completes its execution enters the "terminated" state and the load balancer destroys it.

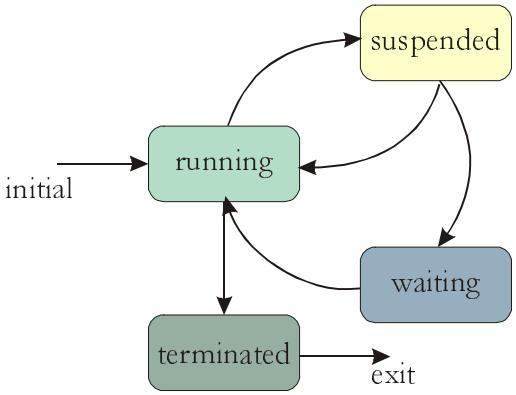


Figure 4: Task state diagram.

4.3. Object Memory

Every task in the NIP abstract parallel system may access objects in the memory by executing methods on them. There is no need for tasks to synchronise their method calls on objects as the object memory takes care of synchronisation issues while trying to insure the maximum parallelism in memory operations. The memory model follows the UFO semantics and allows any number of tasks to call methods on an object at the same time if the methods do not change the state of the object. However, the memory model guarantees that only one task at a time is allowed to change the state of an object. The design and implementation of the memory model is discussed in detail in section 5.4.

4.4. NIP on Distributed Memory Systems

Every node in the distributed memory parallel system (e.g., all the workstations in a network of workstations) runs an instance of a *NIP Node*. A NIP Node consists of a collection of components, each one assigned specific duties. The components of all the NIP Nodes in the system communicate with each other by exchanging messages. The components are:

- The *Node Manager* component directs the operations of the rest of the components in a NIP Node. It routes the messages received by other NIP Nodes to the appropriate component and it maintains NIP system and node specific information (e.g., list of NIP Nodes, number of processors available on the current node, etc.).
- The *Load Balancer* is responsible for maintaining information about the tasks running on the current NIP Node. It uses the information to make decisions on whether more tasks

are required on the NIP Node or not. The Load Balancer is described in detail in section 5.4.

- The *DSM Manager* is the component responsible for managing the local memory of the NIP Node and it is described in section 5.4.
- The *Communicator* is a wrapper around the underlying communication layer that is used.

All the NIP Nodes in the system collaborate to execute a program. On every NIP Node, there is a limit on the number of tasks that may be executed in parallel but the number may not be the same for all NIP Nodes. Usually, the limit is equal to the number of processors available on a node.

5. THE NIP EXECUTION MODEL

On every NIP Node, the Load Balancer component deals with the creation and execution of tasks. The set of Load Balancer components in the parallel system co-operates with each other in order to try to ensure that nodes always have tasks to execute. In this section, a novel variant of the *lazy task creation* [9] technique is introduced, the task execution mechanism is described, and the dynamic load balancing scheme used by NIP is presented.

5.1. Lazy Task Creation

Lazy task creation attempts to reduce the cost of creating tasks for concurrent execution. In [9], a technique is described where every task maintains a queue of *continuations*. A continuation represents a *lazy future call*, a piece of computation that can be executed in parallel. As computational resources become available, a continuation is stolen and a new task is created. When the task requires the result from a continuation, it removes the continuation from the queue and executes it *inline*. The task creating a continuation is called *producer* while a processor stealing a continuation is called *consumer*. The continuation queue is guarded by a lock in order to avoid simultaneous access by the producer and consumer. Mohr et al show that it is beneficial to use continuations in order to represent potentially parallel pieces of computation that may be converted to tasks at runtime rather than creating tasks for all of them.

An important issue for efficient parallel computation is the cost of the inline execution of the continuation, which should not be much greater than the cost of executing the same computation in a purely serial environment. Therefore, in a parallel system using a lazy task creation scheme, it is very important that the cost of creating continuations and executing them inline is minimised.

Analysis of the existing lazy task creation technique identified three causes of inefficiency:

1. The continuation queue is maintained on a heap and, therefore, the creation and destruction of continuations are expensive operations because memory must be allocated/freed on a heap.
2. For iterative computations where each iteration can be executed in parallel, the number of continuations created equals the number of iterations. As the creation of each continuation has a finite cost, this is a large overhead. For example, if a function is mapped onto an N-element vector, then N continuations will be created.
3. The queue of continuations is local to every task. When a processor requires a task to execute, it searches the queues of the tasks on the other processors in a round-robin

fashion until it finds a non-empty one. When accessing a queue a lock must be acquired which introduces significant overhead. The cost of these lock operations has been stated as the main performance disadvantage of lazy task creation in [10].

5.2. NIP Lazy Task Creation

In NIP, a novel variation of the lazy task creation technique is used. The NIP Lazy Task Creation has a simpler and more portable design than the original technique. In NIP Lazy Task Creation, tasklets are used as the representations of the potentially parallel parts of the computation. The new technique reduces the effects of the sources of inefficiency identified in the previous section:

1. Tasklets are allocated in the stack frame of the task that creates them, as in the previous section, but the tasklet queue is not allocated on a heap. Instead, the tasklets are directly linked and the Load Balancer only maintains pointers to the first and the last tasklet in the queue. This reduces the memory allocation/deallocation costs, although when new tasklets are created the links must still be updated but this is a relative small cost.

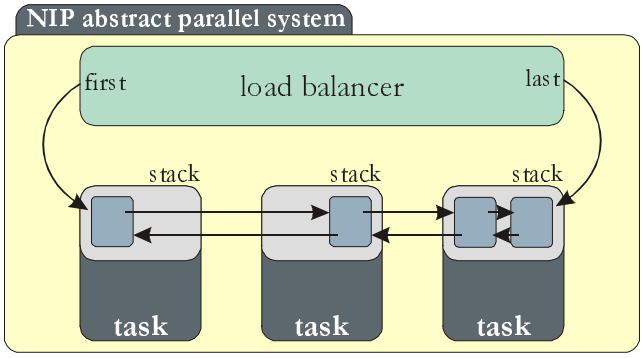


Figure 5: Tasklet queue.

2. Only one tasklet is created for a complete iterative computation (e.g., a for-loop) rather than one for each iteration. Once the tasklet is created, the iterations can be executed serially by the task that created the tasklet, while one or more iterations can be *stolen* by the Load Balancer for execution elsewhere. For example, consider a program in which a function *f* is mapped onto each element of a vector. In pseudo-code, a serial version of the algorithm would be as shown in Figure 6e. In the NIP version (again in pseudo-code), an instance of the `ParallelMap` class (Figure 6a) is constructed. This has as variables the function, the vector, and an index variable which, as will be seen, is shared between the inline code and the task-stealing code. The class constructor initialises the tasklet, publicises it to the Load Balancer by adding it to the local tasklet queue (`activate()`), and then executes the loop inline (Figure 6b). Once the loop is completed the tasklet is removed from the local queue and waits for the results of any stolen iterations to be returned. The Load Balancer can steal iterations by calling the `stealIteration()` method (Figure 6c); a stolen iteration is turned into a task which can be exported for parallel execution. The inline execution and Load Balancer can never execute the same iteration because the critical section in the `stealIndex()` method (Figure 6d) guarantees unique manipulation of the `_index` variable. As an optimisation, to increase the granularity of tasks, more than one iteration can be stolen at a time by the Load Balancer. These "iterative tasklets" are important for reducing costs in a language like UFO which directly supports iteration. However, in languages which do not directly support iteration and offer solely recursion (e.g., most functional languages), techniques for transforming recursion into iteration [11] can still allow this optimisation to be

```
class ParallelMap
{
public:
  ParallelMap(Vector, Function);
  Task configure();
Private:
  int      stealIndex();
  Vector   _vector;
  Function _function;
  int      _index = 0;
}
```

(a)

```
// create task
Task ParallelMap::stealIteration()
{
  tmpi = stealIndex();
  if (tmpi == -1)
    return nothing
  else
    return newTask(_vector,
                   _function,
                   tmpi);
}
```

(c)

```
// execute loop inline
ParallelMap(Vector v, Function f)
{
  _vector   = v;
  _function = f;
  // add tasklet to tasklet queue
  activate();
  while (true)
  {
    tmpi = stealIndex();
    if (tmpi == -1)
       break;
    else
       _vector[tmpi] =
             f(_vector[tmpi]);
  }
  // wait for the results of any exported tasks
  wait();
}
```

(b)

```
int ParallelMap::stealIndex()
{
  startCriticalSection();
  if (_index == _vector.length())
then
    // No work left. Remove tasklet from queue
    deactivate();
    stolen = -1;
  else
    stolen = _i;
    _index++;
  endCriticalSection();
  return stolen;
}
```

(d)

```
map(Function f, Vector v)
{
  for int i = 1 to v.length()
    v[i] = f(v[i]);
}
```

(e)

Figure 6: Pseudo-code for lazy task creation.

exploited. The C++ version of the `ParallelMap` class presented in Figure 6 can be found in the appendix.

3. The tasklet queue is guarded by a lock but a task only needs to acquire this lock when it adds/removes a tasklet to/from the queue. Each tasklet retains a private lock, which must be acquired by the task when it removes part of the computation in order to execute it inline. The Load Balancer needs to acquire both locks when it creates a new task from a tasklet. First, it needs to locate a tasklet to use, so it must acquire the queue's lock. Then, it needs to acquire the lock—private to the tasklet—in order to steal part of the computation. This 2-level locking scheme allows the Load Balancer to create a new task without blocking the rest of the tasks from executing their tasklets inline. Furthermore, on a multiprocessor node, the tasks do not block each other when accessing their tasklets.

4. When the Load Balancer is active, it cannot be interrupted by a task. Therefore, the lock operations can be optimised by using a boolean variable instead of a mutex as the tasklet's private lock, providing the target platform is composed from uniprocessor

workstations where the memory operation of setting the value of a boolean variable is atomic.

5. Finally, tasklets can be designed to be reusable. So, once a tasklet is created it may be used many times in different parts of the computation.

The NIP lazy task creation technique has been implemented in C++ for portability. This differs from earlier schemes (e.g., [9]) which are supported by a customised compiler. However, the performance results given in section 6.1 show that the efficient NIP scheme for iterative computations reduces the need for highly optimised lazy task creation mechanisms.

5.3. Task Execution

As mentioned in section 4, a task is the unit of execution in the NIP abstract parallel system. However, in most of the operating systems, the unit of execution is either a *process* or a *thread*. In view of the lower overheads associated with the creation, switching, and termination of threads on the distributed systems that NIP targets, threads were preferred over processes as the execution medium for tasks. The Load Balancer component is responsible for the creation, suspension, and termination of threads.

When a task enters the "suspended" state, the Load Balancer suspends the execution of the thread, which is resumed when the task moves to the "running" state. There three reasons a task may become "suspended":

- A method is called on an object changing its state while other tasks are executing methods in parallel on the same object. The thread executing the task is suspended until the method call is allowed to proceed.
- A method is called on an object but the internal data of the object is not available in the local memory of the NIP Node. The thread executing the task is suspended until the data is fetched from another NIP Node.
- The task had created a tasklet and now the result from the tasklet is required. The tasklet could have been executed sequentially at this point but the Load Balancer had already converted it to a new task. The thread executing the task waiting for the result is suspended until the new task finishes.

The Load Balancer maintains information about the state of all the tasks on the NIP Node in order to decide whether new tasks are required on the NIP Node or not.

5.4. Dynamic Load Balancing

The set of Load Balancer components in the parallel system co-operate with each other in order to try to ensure that nodes always have tasks to execute. Every Load Balancer implements a simple algorithm to keep its local NIP Node busy with tasks. The algorithm utilises information on tasks and tasklets that the Load Balancer maintains for the NIP Node.

If there are no tasklets when a new tasklet is added, the Load Balancer broadcasts a notification message to the rest of the NIP Nodes. When the last tasklet is removed, a notification message is also broadcasted. Every Load Balancer component maintains a *tasklet availability* vector with a boolean entry for each of the other NIP Nodes in the system. The boolean entry indicates whether the corresponding NIP Node has any tasklets or not, and it is updated upon the receipt of a notification message. When a NIP Node runs out of tasklets

and there are no tasks in the "waiting" state, the Load Balancer checks the tasklet availability vector and sends a request for tasks to a node with a non-empty tasklet queue. The vector is searched in a round-robin fashion, always starting with the NIP Node entry that follows the current NIP node. If a request for tasks is made to a NIP Node that has an empty tasklet queue (as the request was sent before the "no more tasklets" notification message was received) the request is denied.

The semantics of the Load Balancer in the NIP abstract machine are realised with the collaboration of all the Load Balancer components.

6. DISTRIBUTED SHARED MEMORY MODEL

As described in section 3, the target architecture for NIP is a distributed memory parallel machine that does not directly offer shared memory (e.g., a network of workstations). A system providing global access to objects was required for NIP as a task can be executed on any node and it may require access to objects created on other nodes (section 4).

It would have been possible to utilise a conventional, general, all-in-software Distributed Shared Memory (DSM) system to implement the memory model required for NIP [12-19]. However, it was believed that a NIP-specific DSM system, the NIPDSM, could support the semantics of the NIP computational model more efficiently than conventional DSM systems. The NIP computational model imposes certain restrictions on the way objects can be accessed (section 4) and NIPDSM exploits these restrictions—as is described in the rest of this section—in a way that cannot be achieved by a conventional DSM system.

In this section, an architectural overview of the NIPDSM is presented and the motivation behind the design decisions is explained. The memory consistency protocol used in NIPDSM and the way caching is used to improve performance are described. The aspects of the NIPDSM system that exploit the NIP object access semantics are highlighted throughout the section.

6.1. NIPDSM Architecture

Most conventional software-based DSM systems provide a *single address space* view to parallel applications (e.g., [12, 13]). A parallel application running on a distributed memory machine consists of a collection of threads or processes that use read and write operations on virtual addresses—in the same manner as a serial program.

This memory model is not ideal for parallelism as interleaving of read and write operations are allowed without any restrictions, although programmers serialise memory operations using *memory lock* operations. The DSM provides the memory lock operations but it is still the responsibility of the parallel application programmers to use them correctly in order to protect shared memory regions from simultaneous accesses. Failure to do this often results in software errors which are very hard to discover.

NIPDSM investigates an alternative approach, using the semantics of the object memory of the NIP abstract system, where it is the memory that deals with serialisation in memory access rather than the application. The NIPDSM provides an *object view* of the distributed memory; the unit of sharing is the object. The ways in which objects can be accessed by the parallel application are restricted (section 4). There are a few similarities between the NIPDSM object space, the *tuple space* of Linda [20], and the data-structured DSM systems like Emerald and Clouds [19, 21] but there are significant differences as well, as will be described.

The NIPDSM is organised as a collection of objects which are shared amongst the participating NIP nodes. A parallel application cannot directly access memory locations but instead it calls methods upon objects. Depending on whether or not a method updates the state of the object, it is characterised as a *read method* (a *function* in UFO) or as a *write method* (a *procedure* in UFO). An object with only read methods is considered to be *immutable* while an object with at least one write method is considered to be *mutable*. NIPDSM meets the UFO semantics as an object can have multiple concurrent readers but only one writer.

Synchronisation of simultaneous accesses to a mutable object is achieved automatically via the use of a lock which is private to that object. Therefore, there is no need for the application to deal with synchronisation. Nevertheless, an interface to the private lock of each object is available in order to support optimisations by the UFO compiler, when it is more efficient to wrap contiguous method calls on the same object within a single pair of lock/unlock operations.

6.2. Object Replication and Consistency Model

Objects in NIPDSM can be cached in multiple nodes; consequently, provision must be made for their state to be kept consistent across the nodes. Many *consistency models* have been proposed in the literature [13, 22-25] and are used in conventional DSM systems. The choice of the consistency model for the NIPDSM was important not only because of the performance implications but also because the semantics of the NIP computational model have to be met. It was decided that the *entry consistency* model [13] was close to the requirements of the NIPDSM and a novel variation of the original model optimised for NIP was designed and implemented.

Entry consistency is a relaxed model—it defers memory updates to/from replicated regions until they are necessary. Like most relaxed models, entry consistency requires that applications follow certain rules in the way in which they access the shared memory in order to guarantee *strict consistency* semantics in the way the memory operates—a read memory operation always returns the most recent value. An application must associate any shared variables with synchronisation objects, which are used to define critical sections. The shared variables must only be accessed within these critical sections. The burden of defining the critical sections and associating the shared variables with synchronisation objects falls to the application programmer [26].

The semantics of the NIP computational model allow us to simplify the model as there is no need for critical sections at the application level. Every mutable object has a lock which is used whenever a method is applied to the object. This lock provides the synchronisation required when accessing the object's state. Additionally, as will be described, the same lock is used for consistency-related operations ensuring that the object's state is always kept up-to-date. In most conventional DSM designs, memory consistency related operations require the use of separate locks; one used by consistency-related operations and one for synchronisation-related operations.

6.3. Managing Consistency

In this section, the NIPDSM approach in maintaining the consistency of the state of the objects is described. An object in the NIPDSM is always assigned a single *Manager Node*. The Manager Node maintains information about the object and is responsible for satisfying cache requests from other NIP Nodes. When an object is cached on a node, the node becomes either a *Read Proxy Node* or a *Write Proxy Node* for the object. The following rules apply:

- The Manager Node of an object allows the execution of tasks local to that NIP Node which call write and/or read methods on the object when there are no proxies for that object.
- The Manager Node allows the execution of read methods on an object if Read Proxies exist for that object.
- A Write Proxy allows the execution of local tasks that call write and/or read methods on the object.
- A Read Proxy may allow the execution of local threads that call only read methods on the object.
- A node can be a Manager Node, a Read or Write Proxy for any number of objects but can never be more than one of these at the same time for a particular object.
- An object in the NIPDSM always has one Manager Node and it may have zero or one Write Proxy, or zero or many Read Proxies at any given time.

When a task running on a NIP Node requires access to an object and that NIP Node is not the Manager Node or a Proxy for the object, the task is suspended and the object's Manager Node is notified. When a copy of the object arrives from the Manager Node, the node becomes a Read or a Write Proxy, depending on the type of the method called (read or write). A task running on a Read Proxy for a particular object may also have to be suspended if it calls a write method on that object. The node must first become a Write Proxy for the object before the task is allowed to continue. The decision-matrix of Figure 7 shows all the possible actions NIPDSM may take on encountering a method call on an object. The matrix also shows how NIPDSM deals with multiple tasks running on a NIP Node requiring access to the same object at the same time.

Node type for the object	Lock type	Type of call on the object	
		Read method	Write method
Manager Node	Free	Execute	Execute
Manager Node	Read	Execute	Suspend until unlocked (queue)
Manager Node	Write	Suspend until unlocked (queue)	Suspend until unlocked (queue)
Read Proxy	Free	Execute	Suspend (send write proxy req.)
Read Proxy	Read	Execute	Suspend (send write proxy req.)
Write Proxy	Free	Execute	Execute
Write Proxy	Read	Execute	Suspend until unlocked (queue)
Write Proxy	Write	Suspend until unlocked (queue)	Suspend until unlocked (queue)
Not cached	N/A	Suspend (send read proxy req.)	Suspend (send write proxy req.)

Figure 7: Decision-matrix for locking an object.

When the Manager Node receives a request from a node to become a Write or a Read Proxy, it acts according to the decision-matrix of Figure 8. As the matrix shows, the consistency model requires that in some cases the active Proxies must be invalidated before a proxy request can be satisfied.

Lock type	Proxy(ies)	Proxy request	
		Read	Write
Free	N/A	Satisfy	Satisfy
Read	No Proxies	Satisfy	Queue until unlocked
Write	No Proxies	Queue until unlocked	Queue until unlocked
Read	Read Proxies	Satisfy	Queue until invalidated
Write	Write Proxy	Queue until invalidated	Queue until invalidated

Figure 8: Decision-matrix at the Manager Node.

6.4. Implementation

The Manager Node remains the same throughout an object's lifetime. This is in contrast with Bershad's implementation of the entry consistency model [26] where the Manager Node—in NIPDSM terms—was the node with write permission to the object and a distributed directory scheme was used to keep track of the Manager Node. In NIPDSM, the overhead associated with maintaining the distributed directory is avoided in order to simplify the implementation while optimisations are introduced that reduce the number of messages exchanged.

Every object is assigned a unique object ID, the *NIPDSM Pointer*, when it is created. The NIPDSM Pointer also provides access to an object's methods acting like a virtual memory pointer and it is used by the NIPDSM system in all locking and caching operations.

Every NIP Node maintains a table, known as the *NIPDSM Virtual Object Table*, with information on all the objects currently in its local memory. The NIPDSM Pointer is also used as an index into the table. The information found in the table includes:

- A pointer to the location of the object in the local memory
- The state of the object (cached and/or locked)
- The pending lock requests
- The node type for the object (Manager Node, Read Proxy, Write Proxy)
- The number of read locks (if the object is read locked).
- If the NIP Node is the Node Manager for the object, a list of the Proxies is also maintained.

Unlike most conventional DSM systems, NIPDSM does not depend on the operating system to detect memory accesses and trap cache misses. An object can only be accessed via method calls and only via the use of a NIPDSM Pointer. The advantages of this approach are:

- The NIPDSM system is portable, as no operating system specific calls are required.
- Conventional DSM systems use the underlying operating system's page fault mechanism in their implementation. The page fault mechanism usually involves traps to the kernel and the invocation of user-defined handlers. NIPDSM uses the lock operations required by the consistency protocol to detect object accesses. These lock operations would have been required even if NIPDSM were to use the underlying operating system's page-fault mechanism.

A compiler can further optimise an object's access time. Once a method is called on an object and the lock is acquired, the local virtual memory address of the object is returned to the calling thread. As the object is locked, it is safe to use the virtual memory address to access the object instead of the NIPDSM Pointer.

Additional techniques are introduced that attempt to improve the performance of the NIPDSM:

- When a method call cannot be executed immediately, it is queued by the Node Manager or the Proxy Node. It is possible to rearrange the order of the queued method calls in an effort to achieve greater parallelism without losing the entry consistency semantics.

- The performance of the caching mechanism can be enhanced by creating groups of objects. When an object is accessed by a remote NIP Node, a group of objects will be cached to that NIP Node. The approach eliminates the problem of spatial locality that object-based DSM system exhibit. Furthermore, groups of objects can be used for dynamic data structures to improve the performance of operations on them.

7. PERFORMANCE RESULTS

In this section, preliminary performance results from the execution of the prototype implementation of NIP are presented. The Newcastle Affordable Parallel Platform (eight PentiumII 233MHz workstations with 64MB of memory that are interconnected by a 100Mbps Fast Ethernet network) was used as the test-bed. Each of the workstations was running an instance of the Linux operating system (RedHat 5.1, kernel 2.0.35).

The test is the implementation of the UFO example shown in Figure 1 (section 2.1) where a function *f* is mapped on the elements of a vector sequentially and then in parallel using NIP. The sequential program was implemented in standard C while the parallel version was implemented in C++ using the implementation of the `ParallelMap` tasklet class (section 5.2). The egcs 1.1b compiler was used for the compilation of the two versions of the program.

The costs of using the tasklet were measured by comparing the timing results from the execution of the sequential and NIP versions of the program. The overhead incurred because of the creation and destruction of the tasklet and the cost per iteration of using the tasklet are presented in Figure 9.

Tasklet creation and destruction	1.364 usecs
Overhead per iteration	0.16 usecs

Figure 9: Tasklet-related costs.

The `ParallelMap` tasklet only allowed one iteration to be stolen at a time. An optimised version of the tasklet was also tested that exposed a group of iterations at a time and, therefore, increasing the granularity of the tasks created. The number of iterations stolen was calculated at runtime as *n=vector size / (number of nodes * 2)*. The following three tables present the speedups achieved over the sequential version of the program when using different vector sizes and different granularities of the mapped function on 2, 4, and 8 workstations. The results show that NIP performs reasonably well even for small granularities.

		Function granularity (msecs) – Single iteration stolen					Function granularity (msecs) – Group of iterations stolen				
		0.03	**0.3**	**1.5**	**3.02**	**6.04**	**0.03**	**0.3**	**1.5**	**3.02**	**6.04**
Vector size	**100**	0.67	0.91	1.14	1.37	1.78	0.66	1.30	1.60	1.85	1.90
	200	0.80	0.96	1.17	1.38	1.81	0.77	1.39	1.85	1.90	1.96
	500	0.83	0.98	1.19	1.40	1.82	0.44	1.31	1.21	1.51	1.72
	1000	0.91	1.00	1.19	1.40	1.83	1.26	0.93	1.18	1.49	1.67

Figure 10: Speedup on 2 nodes.

		Function granularity (msecs) – Single iteration stolen					Function granularity (msecs) – Group of iterations stolen				
		0.03	**0.3**	**1.5**	**3.02**	**6.04**	**0.03**	**0.3**	**1.5**	**3.02**	**6.04**
Vector size	**100**	0.34	0.91	1.30	2.08	1.87	0.48	1.38	2.32	3.00	3.25
	200	0.80	1.01	1.47	1.23	1.79	0.56	1.80	2.89	3.68	3.80
	500	0.75	1.05	1.13	2.20	3.45	0.97	2.37	3.71	3.77	3.82
	1000	0.91	1.06	1.21	2.21	2.81	0.64	1.16	2.31	3.01	3.42

Figure 11: Speedup on 4 nodes.

		Function granularity (msecs) – Single iteration stolen					Function granularity (msecs) – Group of iterations stolen				
		0.03	0.3	1.5	3.02	6.04	0.03	0.3	1.5	3.02	6.04
Vector size	100	0.39	0.95	2.01	3.29	5.74	0.96	1.36	3.11	4.25	5.26
	200	0.74	1.07	1.99	3.57	5.60	0.59	1.98	4.64	5.09	6.14
	500	0.81	1.11	2.20	3.76	6.65	1.00	3.10	6.92	7.32	7.50
	1000	0.90	1.17	2.38	3.83	6.64	1.34	4.45	7.31	7.21	7.52

Figure 12: Speedup on 8 nodes.

8. CONCLUSIONS AND FURTHER WORK

In this paper, the design of the NIP system for executing parallel UFO programs has been described. UFO combines elements of both object-oriented and functional programming, marrying the implicit parallelism of functional languages with the encapsulation of state manipulation offered by object-oriented languages. Whilst this paper focuses on UFO as the high level language, the issues, design and implementation are applicable to other similar parallel languages, and to any others which offer some of the features found within UFO (e.g., iteration, encapsulation of state within objects). The paper makes the following contributions:

- It investigates the issues and options in the design of systems for executing parallel functional plus object programs on distributed memory parallel systems.
- It introduces a novel and efficient method of lazily extracting parallel work from loops.
- It describes the design of an all-in-software object-based DSM that uses an optimised version of the entry consistency protocol.

There are plans to extend NIP in a number of directions. First, the behaviour of a wider range of parallel programs will be analysed. A set of advanced caching features are also being designed to improve the efficiency of the NIPDSM by copying and caching groups of objects rather than single objects. Finally, we are investigating the mapping of other high level languages onto NIP; the initial focus is on extracting implicit parallelism from Java programs.

This work has investigated the design of an efficient runtime system to support implicit parallel programming. Implicit parallel programming has significant advantages over current mainstream parallel programming methodologies such as MPI and PVM. Programmer productivity is increased by reducing the amount of information on the parallel execution strategy which must be included in the program. It also results in greater program portability as the compiler and runtime have more flexibility in terms of how the program can be most efficiently executed on a particular architecture. As a consequence, however, the runtime system has to be more complex as it takes the responsibility from the programmer in lazily creating work as required to keep the system busy; copying and caching data whilst maintaining coherence; and, load balancing. The paper has described the NIP approach in each of these areas. The results of the investigation, including the preliminary results presented here, show this approach to be viable and worth further investigation.

9. REFERENCES

1. Sunderam, V.S., PVM: A Framework for Parallel Distributed Computing. Concurrency: Practice and Experience, 1990. 2(4)(December 1990): p. 315-339.
2. Forum, M.P.I., MPI: A Message-Passing Interface Standard. International Journal of Supercomputer Applications and High Performance Computing, 1994. 8(3/1).

3. Sterling, T., P. Mesina, and P.H. Smith, Enabling Technologies for Petaflops Computing. 1995: MIT Press.

4. Henson, M.C., Elements of Functional Languages. 1987: Blackwell Scientific Publications.

5. Hammond, K. Parallel Functional Programming: an Introduction. in PASCO'94 - Conference on Parallel Symbolic Computation. 1994. Castle Hagenburg, RISC-LINZ, Austria: World Scientific.

6. Darlington, J. and M.J. Reeve. ALICE: A Multiple-processor Reduction Machine for the Parallel Evaluation of Applicative Languages. in FPCA'81. 1981.

7. Watson, I., et al. Flagship: a Parallel Architecture for Declarative Programming. in 15th Annual ACM Symposium on Computer Architecture. 1988.

8. Sargeant, J., United Functions and Objects: An Overview, . 1993, Dept. of Computer Science, University of Manchester.

9. Mohr, E., D.A. Kranz, and R.H.J. Halstead, Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. IEEE Transactions on Parallel and Distributed Systems, 1991. 2(3): p. 264-280.

10. Kaser, O., et al., EQUALS - a fast parallel implementation of a lazy language. Journal of Functional Programming, 1997. 7: p. 183-217.

11. Burstall, R.M. and J. Darlington, Some transformations for developing recursive programs. Journal of ACM, 1977. 24(1): p. 44-67.

12. Keleher, P., et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. in 1994 Winter USENIX Conference. 1994. San Francisco, CA, USA.

13. Bershad, B.N. and M.J. Zekauskas, Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, . 1991, School of Computer Science, Carnegie Mellon University.

14. Liang, W.-Y., C.-T. King, and F. Lai. Adsmith: An Efficient Object-Based Distributed Shared Memory System on PVM. in 2nd International Symposium o Parallel Architectures, Algorithms, and Networks (I-SPAN 96). 1996. Beijing, Peoples R. China: IEEE Computer Soc. Press.

15. Scales, D.J., K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. in 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII). 1996. Cambridge, MA.

16. Johnson, K.L., M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. in 15th Symposium on Operating Systems Principles. 1995. Colorado, USA.

17. Carter, J.B., J.K. Bennett, and W. Swaenepoel. Implementation and Performance of Munin. in 13th ACM Symposium on Operating Systems Principles. 1991.

18. Scales, D.J. and M.S. Lam, An Efficient Shared Memory Layer for Distributed Memory Machines, . 1994, Computer Systems Laboratory, Stanford University: CA 94305.

19. Jul, E., et al., Fine-Grained Mobility in the Emerald System. ACM Transactions on Computer Systems, 1988. 6(1): p. 109-133.

20. Ahuja, S., N. Carriero, and D. Gelernter, Linda and Friends. Computer, 1986. 19(8): p. 26-34.

21. Ramachandran, U. and M.Y.A. Khalidi, An Implementation of Distributed Shared Memory. Software Practise and Experience, 1991. 21(5): p. 443464.

22. Keleher, P., A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. in 19th Annual Internation Symposium on Computer Architecture. 1992.

23. Iftode, L., J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. in 8th Annual ACM Symposium on Parallel Algorithms and Architectures. 1996.

24. Blumofe, R.D., M. Frigo, and C.F. Joerg. Dag-Consistent Distributed Shared Memory. in 10th International Parallel Processing Symposium. 1996. Honolulu, Hawaii.

25. Mosberger, D., Memory Consistency Models, . 1993, Dept. of Computing Science, University of Arizona.

26. Bershad, B.N., M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. in IEEE COMPCON Conference. 1993.

10. APPENDIX

The lazy task creation mechanism in NIP is illustrated in more detail than in the body of this paper using as an example a function that maps another function over each element of a vector. A UFO version of this program was shown in Figure 1. A serial, C++ template implementation of this function is shown in Figure 13.

```
template <class T>
void map(vector<T>& v, T (*f)(const T&))
{
   for(int i = 0; i < v.size(); i++)
      v(i) = f(v(i));
}
```

Figure 13: A serial template function to map a function over a vector.

A parallel version of the map function, as implemented in NIP, is shown in Figure 14 to Figure 18. First, we need to design a class (`ParallelMap`), which inherits the provided abstract class `NIPTasklet`. Objects of type `ParallelMap` are used where the serial version of the map function would have been used. The interface of a parameterised version of `ParallelMap` is shown in Figure 14.

```
template<class T>
class ParallelMap : public NIPTasklet
{
public:
   ParallelMap() : NIPTasklet(NIPTasklet::asynchronous, &execute) {}
   virtual ~ParallelMap() {}

   void        finish(NIPTask&);
   static void execute(NIPTask&);
   bool        configure(NIPTask&);
   void        operator()(vector<T>*, T (*)(const T&));
private:
   int        _i;
   vector<T>* _v;
   T        (*_f)(const T&);
};
```

Figure 14: The interface of a user-defined tasklet.

Objects of type `ParallelMap` maintain a variable `_i`, which is used for counting the elapsed iterations. An instance of the `ParallelMap` class may be used where a loop would have been used. The three virtual methods inherited from `NIPTasklet` must, now, be defined (Figure 15, Figure 16, Figure 17).

```
template<class T>
bool ParallelMap<T>::configure(NIPTask& task)
{
   startCriticalSection();
   int tmpi = _i++;
   endCriticalSection();

   if (tmpi < (*_v).size())
   {
      task.packet().pack(tmpi);
      task.packet().pack((*_v)[tmpi]);
      task.packet().pack((long)_f);
      return true;
   }
   else
   {
      deactivate();
      return false;
   }
}
```

Figure 15: The `configure` method of the `ParallelMap` template class.

The `configure` method is always executed in the work stealer's high priority thread. It is called when a new task must be created. In this case, the task is configured to contain the necessary information for the execution of only one iteration, which is now considered to be stolen. The information stored into the `NIPTask` object are the iteration number (`_i`), the value of the i^{th} element of the vector, and the pointer to the function to be applied to that element. The update to the variable _i is guarded by the `startCriticalSection` and `endCriticalSection` calls, which are the interface to the tasklet's lock. (Figure 15)

```
template<class T>
static void ParallelMap<T>::execute(NIPTask& task)
{
   T (*f)(const T&);
   T tmp;
   task.packet().unpack((long&)f);
   task.packet().unpack(tmp);
   tmp = f(tmp);
   task.packet().pack(tmp);
}
```

Figure 16: The `execute` method of the `ParallelMap` template class.

The `execute` static method is called by a worker at a separate thread of control—most often on another node. The information necessary to execute the stolen iteration is retrieved from the `NIPTask` object. At the end of the computation the result is stored back to the `NIPTask` object. (Figure 16)

```
template<class T>
void ParallelMap<T>::finish(NIPTask& task)
{
   T   tmp;
   int i;
   task.packet().unpack(tmp);
   task.packet().unpack(i);
   (*_v)[i] = tmp;
}
```

Figure 17: The `finish` method of the `ParallelMap` template class.

When the parallel execution of the stolen iteration has completed, the `NIPTask` object is passed to the `finish` method that retrieves the result. The work stealer calls the method from within the high priority thread. (Figure 17)

The function operator (`operator()`) is not NIP-related. The loop could have been implemented as a method or as part of the constructor. Although the first alternative is not different from the one shown, the latter would have limited the number of times the instances of the class could be used. Currently, an instance of the class can be used many times. However, if the algorithm was part of the constructor, an instance could only be used once.

```
template<class T>
void ParallelMap<T>::operator()(vector<T>* v, T (*f)(const T&))
{
   _v = v;
   _f = f;
   _i = 0;
   activate();
   while(true)
   {
      startCriticalSection();
      int tmpi = _i++;
      endCriticalSection();
      if (tmpi < (*_v).size())
      {
         (*_v)[tmpi] = _f((*_v)[tmpi]);
      }
      else
      {
         deactivate();
         break;
      }
   }
}
```

Figure 18: The function operator of the `ParallelMap` template class.

Finally, we can use an instance of the `ParallelMap` class—the tasklet—to map a function over the elements of a vector. As we need to be sure that all the elements of the vector have been updated before we proceed, we call the method `wait`. The tasklet could then be reused to map another function over the same or a different vector.

```
#define X 100

double negative(const double& d)
{
   return (-d);
}

double sqrt(const double& d)
{
   return (d * d);
}

int nipMain(int, char*[])
{
   vector<double>      v(X);
   ParallelMap<double> pmap;

   pmap(&v, &sqrt);
   pmap.wait();
   pmap(&v, &negative);

   return 0;
}
```

Figure 19: How the `ParallelMap` class is used.

In order to simplify the task of the UFO compiler in generating new tasklets, the NIP library provides some useful tasklets that can be used directly. For example, Figure 20 illustrates how a simple loop on function *f* can be written using the NIP-provided `NIPLoop` class.

```
...
NIPLoop loop;
loop(start, end, step, f);
...
```

Figure 20: A NIP loop tasklet.