

---

---

# Contents

<b>1</b>	<b>A C++/Tuple-Lock Implementation for Distributed Objects</b>	<b>1</b>
1.1	Introduction	1
1.1.1	Linda! Linda!	2
1.1.2	C++/Tuple-Locks	3
1.2	Tuple-Lock	3
1.3	Using C++	6
1.4	Primitives	6
1.5	Design and Implementation	9
1.5.1	Parallel Virtual Machine	9
1.5.2	Preprocessor Link	9
1.5.3	Shared Object Handler	10
1.5.4	Central Registrar	13
1.5.5	Distributed Processing	13
1.5.6	Replication	14
1.6	Examples	17
1.6.1	One-Way Bridge Problem	18
1.6.2	Jacobi	18
1.6.3	FIFO Queue	21
1.7	Results	24
1.7.1	Startup Costs	25
1.7.2	Tuple-Lock Costs	25
1.8	Advantages and Disadvantages	27
1.9	Conclusion	28
1.10	Bibliography	28



# A C++/Tuple-Lock Implementation for Distributed Objects

GEOFFREY SY

School of Computing  
National University of Singapore  
Singapore  
*geoffrey@comp.nus.edu.sg*

### 1.1 Introduction

Locks act as a synchronization tool that guards the shared memory from being accessed at the same time. Locks treat a block of code (critical section) as one single atomic action. This allows the process to prevent other processes from changing the data in the middle of calculation or update. Locks are usually provided in DSM systems. They come in different forms. Some provide only a very simple locking mechanism that guards the critical section. An example is the single reader/single writer mechanism. The SRSW only allows one process to be in the critical section at any time. There are also more complicated locking mechanisms. Among these are the multiple reader/single writer (MRSW) and the multiple reader/multiple writer (MRMW) schemes. In the schemes above, a distinction is made between a read lock and a write lock. In so doing, more parallelism is hoped to achieve. Since readers don't change the value of a shared memory, then more than one reader are allowed to use the shared memory. Multiple writers are allowed when there would be no concurrency problem for the program. Such programs are called data-race free programs.

Usually, in a typical DSM system, only one of the above schemes is implemented. For instance, if a lock is MRSW, then usually it couldn't be MRMW at the same time or vice-versa. There is a trade-off between the two. For MRSW, it maybe less

efficient since it only allows one writer at a time, but it provides a stricter memory coherence model and avoid concurrency problems. For MRMW, it maybe more efficient by allowing more than one writer at a time, but such model may be more prone to memory coherence error. An example of MRMW is concurrently writing to individual elements of an array. There are also times when the program needs MRSW in one instance and MRMW in another instance.

The problem with locks is not only as to what kind of locks are provided but also when are they acquired or released. Such expression cannot be expressed on the simple parameterless lock we often see. For example, expressions such as: a pop lock acquire to a stack would only be acquired when the stack is not empty or a push lock release would only be successful when the stack is not full is hard if not impossible to express for simple locks. However, simple locks have their merits too for being simple. Some parallel programs may not need such complex types of locks and would be contented with the simpler one. A richer in content form of lock, which can tune in and out from being simple to complex when the need arises, is needed to provide flexibility and efficiency to the program.

Another aspect of locks is in data association. Without associating locks to data either implicitly or through specification, false exclusion and false sharing may occur. False exclusion occurs when processes performing on unrelated data cannot execute in parallel or are excluded from each other because they are guarded by the same lock. False sharing occurs when data are guarded by the same lock even though there is no actual content sharing. However, having multiple locks to guard different shared data may be more efficient in terms of parallelization but may be costlier in terms of maintenance.

In conclusion, the mechanics of a lock would depend largely on the requirements of a program. A parallel programming tool should provide the flexibility on supporting different types of locking mechanisms to benefit the programmer in providing a more efficient DSM system. Since the programmer knows best about its program, it is only right for the system to be tailored to the programmer needs rather than the programmer trying to adapt. Thus, in a way, a more general kind of parallel programming tool is needed to cater the different requirements and needs of a program.

### 1.1.1 Linda! Linda!

A generative memory consistency model is designed to cater to the complex requirements and needs of a parallel program by augmenting the capabilities of a DSM system. Linda is one system that exhibits a generative memory consistency model. Linda uses a tuple space model of parallel programming. Processes coordinate with each other through creating, removing and matching tuples that are stored in the tuple space. The tuple space is a virtual memory visible to every processors in a distributed system, but physically located locally and/or in one or more remote machines. If a process tries to remove or match a tuple that is not present in the tuple space, the process is suspended until the tuple is present.

A tuple contains information in its parameters. This information can be used in solving synchronization issues of a specific parallel problem or model certain types of locks. For instance, in modeling a multiple reader/single writer type of lock, the number of readers and writers can be recorded in the parameters. For the stack problem mentioned previously, a 'Stack, Empty' or 'Stack, NotEmpty' can be placed in the parameters. For a pop to be successful, the process has to match a tuple with parameters 'Stack, NotEmpty'.

Although Linda system provides the framework for flexibility, it has the disadvantage of having to search through the tuple space in order to match tuples. As the number of tuples grow increasingly, the searching delay for matching and locating the right tuple also increases. Although the searching delay can be partially reduced through the use of hashing tables and subspace construction, there are no generally agreed scope rules for tuple operations to be systematically reduced [19].

### 1.1.2 C++/Tuple-Locks

Our method of embedded tuple-locks within shared objects extends the concept of Linda tuple space by providing a fixed location for each tuple. Queries then specify the tuple it is trying to match. Thus, the searching process is greatly reduced though not completely eliminated [24]. Moreover, tuple-locks not only retain the expressiveness of Linda with regards to tuple matching, but also further enhance this expressive power to compensate for the loss of creating and removing tuples dynamically. By allowing not only match, but also modify capabilities on tuple parameters in one single operation, tuple-locks indeed provide a content-rich and flexible set of locks. This enhances the efficiency of distributed programs without the loss of ease to programmers.

By providing flexibility and power to express complex operations, writing tuple-locks can vary from easy to difficult. C++/tuple-lock implementation adds another layer on top of tuple-locks that encapsulate the complexities of deploying them. On one hand, it enables one to write a library of distributed data structures with locks that are tailored to their behavior; thereby, providing efficiency. On the other hand, these structures can then be reused with most, if not all, of the synchronization issues hidden from the programmer. In addition, C++ fits in tuple-locks naturally as scope rules and object to tuple-lock associations are clearly defined.

## 1.2 Tuple-Lock

Unlike Linda tuples, tuple-locks are declared. They are fixed in number and fields, and are associated with a particular object. Tuple-locks are not added to and removed from the tuple space. They merely change their contents. Thus, the location of tuple-locks becomes static.

In defining a tuple-lock, the number and types of contents that it will hold is declared and separated into fields. For instance, tuple-lock A might have three fields. Field one is an integer. Field two is a string and field three is a floating

point number. There are three operations that query and operate on tuple-locks. These are the: InBucket, OutBucket and RdTuple. Each refers to a tuple-lock in a particular shared object, which is regarded as a storage bucket containing data guarded by its tuple-lock(s). The shared object is replicated locally on each node. The three operations match and modify the tuple-lock contents. The InBucket operation also gathers new modifications from the shared object and copy it to the local object while the OutBucket operation flushes out the modifications to allow the new content to be shared by others. Below shows the general structure of the tuple-lock operations.

```
InBucket (BucketID.TupleID, pair, ..., pair)
OutBucket (BucketID.TupleID, pair, ..., pair)
RdTuple (BucketID.TupleID, pair, ..., pair)
```

BucketID.TupleID is the address of the tuple. Thus, as mentioned earlier, searching the location of the tuple is not necessary. The number of pairs is equivalent to the number of fields declared when defining the specific tuple-lock, and each pair operates on a field as indicated by position. A pair has two formats: match | modify or | modify| match. Each match is any expression that gives a value. While a modify has an operation in addition to an expression.

For match, the value is compared against the value of the corresponding field. If it does not match, the query is stored and its thread is suspended. The query is reactivated when there is a change in the tuple-lock contents causing a match; this may require all the waiting requests on the tuple to be searched to see if the new content matches. For modify, the value replaces the field's value. The expression ?var and [?bucket.var] is also allowed. The ?var expression stores the value of the field into a local variable. The [?bucket.var] expression in match indicates the position of an element within the bucket for the modify expression. This allows direct modification to the shared bucket that is stored remotely. The expression can also be a blank one. A blank expression is instantly satisfied.

In a tuple-lock operation, if all the pairs are match | modify, the match of all pairs is first performed. When one match does not succeed, then all match do not succeed. The query is stored and suspended. All match of the match | modify pairs have to be done again when the query reactivates later. When all match succeeds, then the modify of all pairs is performed. For the | modify | match pair, all the modify are first performed then match. In cases when both formats exist in a tuple-lock operation, the match of the match | modify pairs is first performed. Then, all modify are performed, followed by the match of the | modify | match pair. The process that calls the tuple-lock operation is blocked until all field expressions are satisfied. This gives a lock that is flexible to meet the different needs and requirements of a program. Below gives an introductory example of using tuple-locks to implement mutual exclusion:

```
class object_type {
private:
```

```

... some elements ...

public:
    tuple_lock tuple;
    object_type():tuple("int") {};
} object;

```

Process A	Process B
InBucket(object.tuple, 0 — =1)	InBucket(object.tuple, 0 — =1)
... modify object ...	
OutBucket(object.tuple, — =0)	modify object
	OutBucket(object.tuple, — =0)

**Table 1.1.** Mutual Exclusion Using Tuple-locks

In the example above, the instance object is declared to be of type object\_type. It contains a tuple-lock named tuple. The tuple-lock contains one integer field as declared in the object\_type constructor. When both process A and B tried to match the first field of the tuple-lock with the value 0, process A was able to match first. After matching the first field with zero, process A changed this value to 1. This prohibited process B from entering the critical section of the code since it has to match for the value 0. The query of process B is then stored and suspended. The InBucket operation also retrieved the modifications made to object. After modifying the object, process A called the function OutBucket to flush out the modifications and to change the value of the tuple-lock field to 0. This awakened the tuple-lock query and process B was then able to enter the critical section. The InBucket operation enabled process B to retrieve the current value of object. Below is an example of Multiple-Reader/Single-Writer (MRSW) locking mechanism as adopted from [24]:

Readers:

```

InBucket (BucketID.TupleID: 0, $$ +1)
... read ...
OutBucket (BucketID.TupleID: -, $$ -1)

```

Writers:

```

InBucket (BucketID.TupleID: 0 $$ =1, -)
... write ...
OutBucket (BucketID.TupleID: $$ =0, 0)

```

The value - in a match means that the field doesn't matter. In the implementation above, when a reader is reading, a writer is not allowed to write. Multiple readers can enter and read the shared memory, while only one writer can write at a time. Writer starvation is prevented when the writer changes the first field to 1, thereby blocking subsequent readers or writers from entering. Because we assume a distributed environment, even though there are still readers reading, the writer can modify its local copy. The writer waits for the readers to be all finished as indicated by the second field before flushing out the modifications. This allows the writer to modify while waiting for the readers to finish. The function prototypes used in these examples vary slightly from the C++/tuple-lock implementation to be shown in the subsequent sections. These examples serve as an introduction to tuple-lock and are meant for understanding it easily without the hassle of understanding the syntax of the C++/tuple-lock implementation.

### 1.3 Using C++

A C++/tuple-lock model allows a more seamless integration between objects and tuple-locks. When a tuple-lock is declared inside an object, an association is automatically enforced. An InBucket tuple-lock operation retrieves new modifications to its associated object while an OutBucket tuple-lock operation flushes out the modifications to the shared object. Thus, the tuple-lock is then identified through the object name and the tuple-lock name. This design defines the scope and association of a tuple-lock clearly and prevents false sharing.

C++ as an object oriented programming language, can provide another layer on top of tuple-lock to ease and help in distributed programming. As will be shown in the succeeding sections, the programmer can encapsulate all the parallel programming needs inside the class. This class can then be distributed to other programmers without them having to understand the parallel programming codes. They can just simply deploy them in their programs and focus more on other problems. Furthermore, templates capability in C++ allows the user to create generic distributed objects and reuse these objects. For more discussion on C++, please refer to [12] [17] [23].

### 1.4 Primitives

The C++/tuple-lock implementation is encapsulated on a class called GM (global memory). GM requires one string parameter when initialized. This string value associates the current process with the intended distributed group. This allows shared memory objects to be visible only within this group and avoid it to be mixed up with another distributed group that uses the same object name.

GM provides two primitives for task creation and division. This supports the master-slave architecture where the master spawns slaves to do its tasks. Their function prototypes are as follows:

```
spawn (const char *str, [int num]);
id(void);
```

The function `spawn` spawns the process named `str`. The variable `num` specifies the number of times the process is to be spawned. This field is optional and has a default value of 1. GM will select the most appropriate host as to where the process will reside. The function `id` returns the instance number of a process. The instance number of a process is a number relative to the sequence in which the process join the virtual distributed group. The master program would have an instance number of 0 and its slaves would have an instance number that ranges from 1 to the number of slaves present. A process joins a virtual distributed group upon instantiating the class GM and leaves when the instantiated GM object is destroyed.

GM provides two primitives to attach and detach local variables to shared objects. This allows remotely located shared objects to be visible locally. Below shows their respective function prototypes:

```
attach (const char *str, void *ptr, size_t size);
detach(const char *str);
```

The function `attach` attaches a local variable which is pointed by `ptr` to the global shared memory object named `str` which has a size `size`. This tells the system to apply all modification of the shared object to the local variable and vice-versa. The detach operation does the opposite.

GM provides the primitives `in`, `out`, and `rd` to perform the operation `InBucket`, `OutBucket`, and `RdTuple` respectively. Their function prototypes are as follows:

```
in (GM_TUPLE *ptr, match | modify | match, ...);
out (GM_TUPLE *ptr, match | modify | match, ...);
rd (GM_TUPLE *ptr, match | modify | match, ...);
```

The variable `ptr` is the address of the tuple-lock. Tuple-lock is defined inside the object using the data structure `GM_TUPLE`. The data structure `GM_TUPLE` is initialized with a string parameter. The string parameter defines the data types of the fields in the tuple. The data types supported are `int`, `short`, `long`, `float`, `double`, `string` and `char`. For instance, as shown in the next listing, class `object_demo` has one `GM_TUPLE`. The `GM_TUPLE` is instantiated as the variable `tuple`. On the class constructor, `tuple` is initialized with two fields: an integer and a float.

```
class object_demo {
public:
    int value;
    GM_TUPLE tuple;

    object_demo():tuple("int, float") {};
} object;
```

The match | modify | match structure are equations or operations, as described in the tuple-lock. Using the preprocessor, one can easily incorporate variables in the equations or operations. An example on tuple-lock primitives and operation using the class `object_demo` definition above is shown in the next listing.

```

1. GM gm("demo");
2. int x=30;
3. gm.attach ("object", &object, sizeof(object));
4. gm.rd(&object.tuple, | =x, 0.0);

```

In the previous listing, line 1 instantiated the GM class. This enrolls the program into the distributed program named "demo". Thus, shared objects that are defined in this program are only visible to programs that enroll in the distributed program named "demo". In line 2, a local integer variable is declared. The variable is named `x` and is initialized with the value of 30. In line 3, the instantiated class `object_demo` named `object` is attached to the global memory. The first field indicates the global name of the shared object that is "object". The second field indicates the address or location of the local variable that would be linked to the shared object. Through linking the local variable to the shared object, any modifications done to the local variable would be flushed out globally during an out operation and any new modifications flushed out by other programs would be applied to the local variable during an in operation by the program. The third field indicates the size of the local object. In line 4, an rd operation is done to `object.tuple` as indicated in the first field. The second field indicates the operation that would be done. There are two match | modify | match structure that is done to the `object.tuple`. The first one does not have the match parts. It merely sets the integer field to the value of `x` that is 30. The second structure only has the first match part. It compares the value of the float field with the value of 0.0. In this scenario, the value of the float field would first be matched before the integer field would be set to 30. This is because in a match | modify | match structure, all of the first match will be done first before modify, and all the modify will be done first before the last match.

As introduced in tuple-lock, one can retrieve the value of the tuple-lock field through the `?var` operation. The `var` is the local variable to which the value will be placed. For instance, in the next listing, the example is quite similar to the previous listing except in lines 2 and 5. In line 2, another local integer variable called `temp` is added. In line 5, the tuple-lock value of the first field is fetched through the `?temp` operation; while the second field of tuple-lock is ignored.

```

1. GM gm("demo");
2. int x=30, temp;
3. gm.attach ("object", &object, sizeof(object));
4. gm.rd(&object.tuple, | =x, 0.0);
5. gm.rd(&object.tuple, ?temp, -);

```

A direct modification can also be done on the shared object. This is done through the `[?object.var]` operation. The `[?object.var]` would be placed in the match part

and the modification to the location would be placed in the modify part. Both the `?var` and `[?object.var]` operation tells about a location to modify. The difference is that for `?var` operation, it tells about a local location, while the `[?object.var]` tells about a remote location to modify. In line 4 of the next listing, the match part of the tuple-lock's first field is the location of the remote shared object to be modified upon. In the modify part is the value to be placed upon the remote object, which is `x`. Thus, the element "value" of the shared object "object" has a value of 30. However, the local variable `object.value` is not yet changed. This is because the operation is an `rd` operation. Thus, `object.value` remains the same. If it were an `in` operation, the current modification to the shared object "object.value" would be retrieved and placed in the locally linked shared object or the local variable `object`.

```
1. GM gm("demo");
2. int x=30;
3. gm.attach ("object", &object, sizeof(object);
4. gm.rd(&object.tuple, [?object.value] | =x, 0.0);
```

## 1.5 Design and Implementation

### 1.5.1 Parallel Virtual Machine

The GM system is built on top of Parallel Virtual Machine (PVM). PVM is a message-passing platform independent of any language, and allows different processes located in different computers to communicate with each other through messages. PVM also provides spawning of processes to remote computers. Having PVM as its foundation, the system hides these capabilities and applies them in a different paradigm - object with tuple-locks in DSM. The system provides primitives for spawning of processes, process identification, declaring shared memory objects and tuple-lock operations.

The Parallel Virtual Machine (PVM) is a function library to the C language and it uses the message-passing model to facilitate programming in a distributed computing environment. PVM allows different computers systems to be viewed as a single parallel virtual machine. PVM handles all message routines, data conversion, and task scheduling across a network of different incompatible computer systems. Moreover, PVM handles this transparently to the programmer [2].

### 1.5.2 Preprocessor Link

The C++ language is not readily available to the syntax of the tuple-lock. A preprocessor is created as the middle layer between tuple-lock and C++. The preprocessor basically converts tuple-lock command into a string understandable by C++. This then allows C++ to easily pack the string into a network message and send it to the shared object handler. Below shows the general conversion structure.

Before Preprocessor	After Preprocessor
GM.in(&bucket.tuple, 0, "hello")	GM.in(&bucket.tuple, "0, "hello")
GM.in(&bucket.tuple, 0   +5)	GM.in(&bucket.tuple, "0 # +5")
GM.in(&bucket.tuple, 0   +a)	GM.in(&bucket.tuple, "0 # +%s", gm_convert(a))
GM.in(&bucket.tuple, ?a)	GM.in(&bucket.tuple, "?var", &a)
GM.in(&bucket.tuple, [?bucket.a]   +5)	GM.in(&bucket.tuple, "[?var] # +5", &bucket.a)

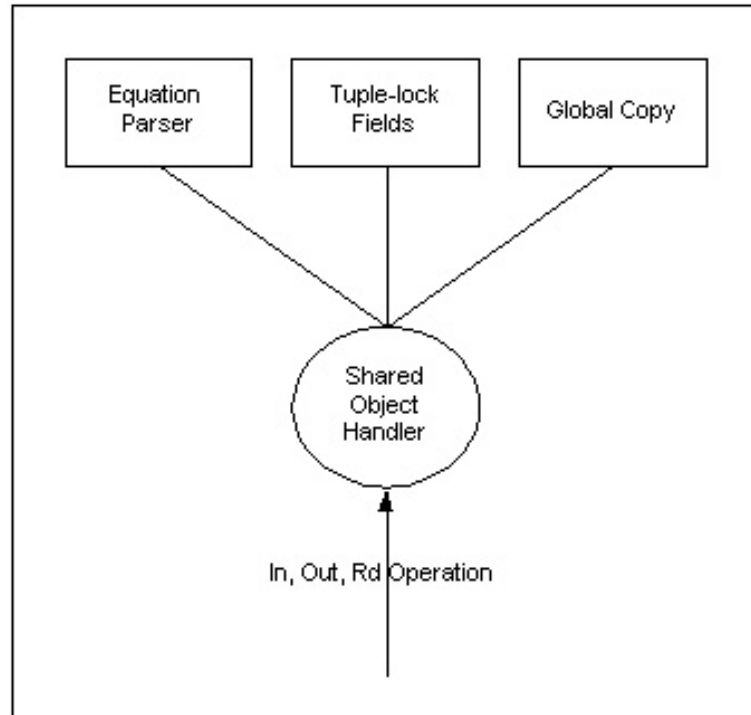
**Table 1.2.** Preprocessor Conversion

Notice that in the first line, the whole tuple-lock commands are placed inside the double quote sign, changing the whole phrase into a string. In the second line, the tuple-lock command of match then modify is shown. Notice that the bar vertical sign that indicates the separation between the match and the modify is converted into the pound sign. This is because the bar vertical sign is used in C++ as an or symbol. In the third line, notice that tuple-lock command is able to take in any kind of variable without knowing if it is an integer, float, double, char, string or etc. This is done through the magic of function overloading in C++ and the function `gm_convert`. This function converts any basic data type into string. In the fourth line, the value of the field is to be placed inside local variable `a`. The conversion involves specifying a new format specification in C++, which is the `?var` symbol, that is not seen in the `printf` family. The `?var` symbol is related to the `&a`, specifying the location where the value of the field is to be written to. The fifth line indicates the location in the global copy, where the modification part will take place. The address is a relative value from the start of the object. For instance, if the locally linked shared object is located at address 800, one couldn't expect that global copy also start at 800. Thus, when the programmer specifies that a modification will be done at 805, it is the relative value that is 5 (805-800) which tells the location. Thus, 5 bytes from the start of the global copy is where the changes will take place.

### 1.5.3 Shared Object Handler

In order to achieve distributed processing of shared objects, each shared object is attached to a process called shared object handler. The shared object handler serves as an interface to the shared object or global copy. It is the process that performs the necessary updates to the shared object. These updates come from programs that made a local modification to the locally linked shared object and flushing out these modifications to make it visible to other programs within the distributed system or program. The tuple-lock fields and values are stored in the shared object handler. These fields are not replicated and their values are not located within the shared object itself. The reason why it is not replicated is because operations on the tuple-lock fields are not done in parallel. Being atomic, they serve as the building

block for synchronization. The shared object handler also handles and evaluates the tuple-lock operations.

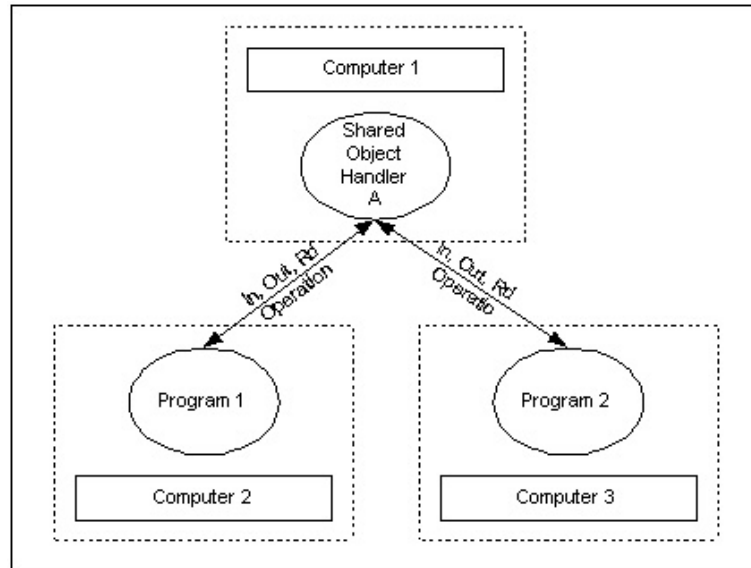


**Figure 1.1.** Shared Object Handler Components

Tuple-lock operations are not evaluated in the local process. Rather, they are sent over to the process handling the shared object in the form of a string and evaluated there. Thus, the process handling the shared object is equipped with the capability to parse and evaluate equations. Equations cannot be parsed and evaluated locally because it lacks the most crucial information, which is the tuple-lock field value. The field value is only visible remotely. Therefore, the equation has to be evaluated remotely.

### Creating Shared Object Handler

When the program calls `GM.attach` to attach a local object to the global copy, what happens is that the program queries the central registrar for its location. If the global copy is present, the central registrar returns the location and the program synchronizes the local object with the global copy. During synchronization, the program collects the current value of the global copy and places the value onto the local the local object.



**Figure 1.2.** Remote Processing of In, Out and Rd Operations

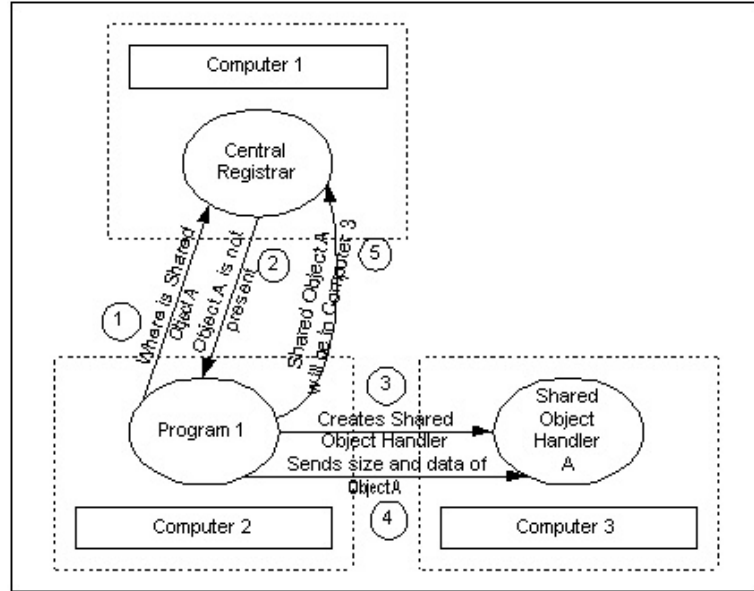
If the global memory is not present or if it is the first to call `GM.attach` to the global memory, the program is tasked on creating shared object and its handler. First, the program creates or spawns the shared object handler using the `pvm_spawn` command. Using this command, the shared object handler will be spawned on the processor that has the lesser load. After which, the program will send the size and data of the local object to the shared object handler. The program assumes that since a global copy of the object is not present or created, having the local object as the first global copy would not violate any synchronization issues.

### Registering Shared Object Handler

After creating the shared object handler, the next step is to register the shared object handler. This allows the shared object to be available to other programs. Registering the shared object is done through sending the location of the shared object handler to the central registrar. When another program wants to attach its local object to the global object, it can attain the information by querying the central registrar. This is discussed in detail in the subsection: "Central Registrar".

### Storing the Location of Shared Object Handler

After a program determines the location of the global copy either by querying the central registrar or by creating the global copy itself, the program stores the location of the global copy in its local memory. This prevents querying the central registrar



**Figure 1.3.** Creating of New Shared Object Handler

every now and then, which only contributes to the network load and the delay in sending and packing network message.

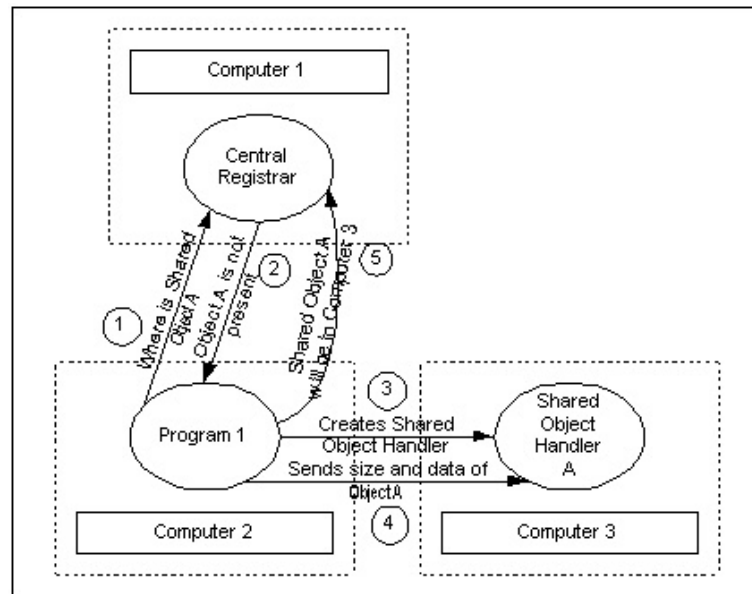
#### 1.5.4 Central Registrar

The system also has a central registrar to keep track of the locations of distributed objects. This allows new processes to obtain the location of distributed objects and their tuple-locks. Processes keep this info on their own local table to avoid querying the registrar every now and then. This design permits object migration to be integrated easily, though implementation of this has been deferred to a later stage.

In Figure 1.4, the process queried the central registrar for the location of the shared object handler of object A. The central registrar responded with the location. The process then synchronizes with the global copy either by receiving new modifications through the in operation, or flushing out modifications through the out operation. The process also sends the tuple-lock operation of the in, out, and rd operation to the shared object handler to be evaluated remotely.

#### 1.5.5 Distributed Processing

Tuple-locks belonging to the same object cannot be stored in a distributed fashion and are therefore incorporated into the process handling the same object. This is



**Figure 1.4.** Querying the Central Registrar for the Location of Shared Object

because tuple-lock operations control modifications to the object. Permitting them to process in parallel might suffer from synchronization problem. Furthermore, if they are distributed far away from the process handling the object, any object modification by the tuple will result from having to send extra messages. However, since tuple-locks belonging to different objects do not concern with each other, they are stored in a distributed fashion. Thus, this allows concurrent accessing and processing of unrelated tuple-lock queries or operations. The clear definition, scope and association of tuple-locks have allowed the system to take advantage of this parallelism.

### 1.5.6 Replication

The system employs replication methods to maintain shared object content. Modifications are stored in the form of diffs. Diffs are data structures that record updates to parts of an object. Unlike for those that synchronize with the global memory through sending the whole shared object, diffs synchronize through recording, sending and storing the modified parts. Thus, only those parts are affected during synchronization. This allows one program to write to one part of the shared object and another program to write to another part without their modifications overlapping each other. Thus, concurrent writes to the same object are allowed. Furthermore, fewer bytes are needed for synchronization. However, in order to ensure that no concurrency problem will exist, the programmer has to ensure that

the program is data-race-free. Data-race-free programs are programs that do not concurrently write to overlapping parts of the same object. Using diffs also reduces the size of the messages sent and contributes to the efficiency of the system.

### Overview of Diffs

The implementation of diffs involves creating twins. A twin is a copy of the shared object. The twin is created in order to compare the state of the shared object before and after a program made some new modifications to the locally linked shared object. By comparing both states, the program determines the changes made to the locally linked shared object. These changes are then collected as diffs. Thus, when the program issues an out operation afterwards, these diffs are flushed out to the shared object handler to be applied to the global copy. When another program issues an in operation, the shared object handler sends the diffs the program hasn't acquired yet.

### Diffs in Out Operation

After a local object is linked to its global copy using the `GM.attach`, a twin is also created and stored locally. Thus, the program is free to modify the locally linked shared object in any way it wishes. This can be either using the object's function, its friends, or a direct member modification that is not encouraged in object-oriented programming. When the program issues an out operation, a comparison would then be made between the locally linked shared object and its twin. Their difference, which is the new modifications made to the object, will be sent as a network message to the shared object handler. The shared object handler then updates the global copy with the diffs received. Meanwhile, the program then performs a twin operation. This is discarding the previous twin and making a copy of the current value of the locally linked shared object. This avoids previously flushed out modification to be sent again.

The comparison made between the twin and the object is done byte by byte. If the bytes were the same, a zero bit would be added to the network message. On the other hand, if the bytes were different, a set bit and the modified byte would be added to the network message. Thus, there is a one to one correspondence between the bits and the size of the shared object. These bits that track the modified byte are placed in the upper half of the network message, and the modified byte for every set bit is placed in the lower half of the network message. When the shared object handler receives the network message, it merely has to understand this format and apply the necessary changes to the global copy.

With every byte of the global copy, an integer is assigned to it. Thus, forming an array of integers. The integer time stamps the byte of the shared object. All the integers start with the value zero. When the shared object handler receives a diff and applies the necessary changes to the global copy, the assigned integers of the modified bytes are time stamped to the next time stamp value. The next time stamp value is determined by finding the present largest time stamp value and

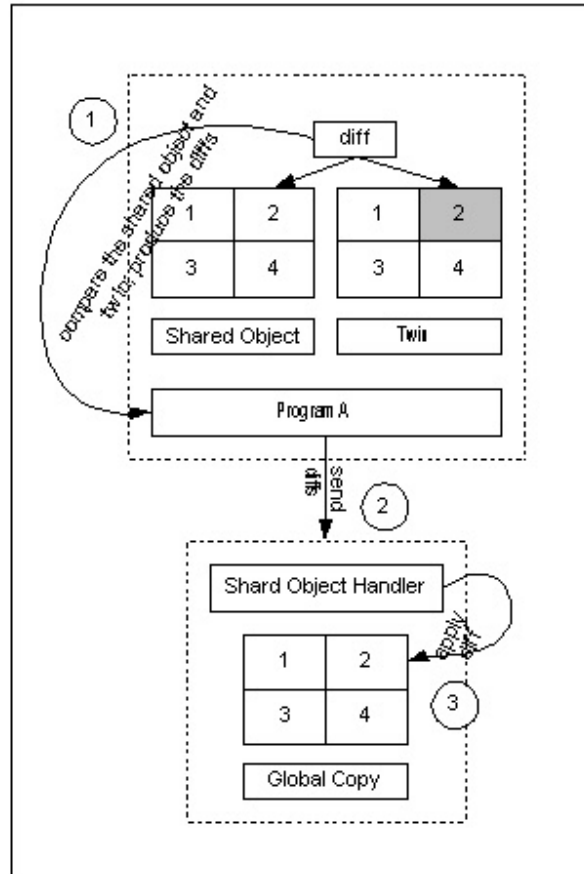


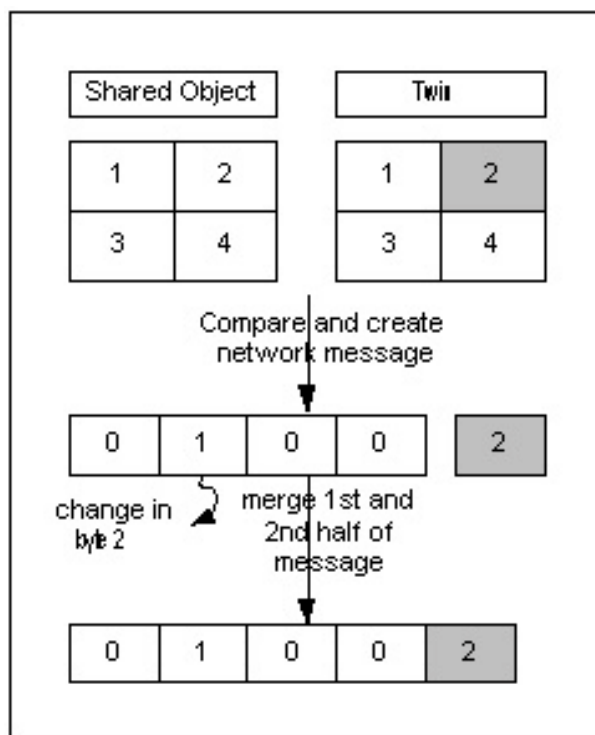
Figure 1.5. Diff Overview

incrementing it by one. The use of these integers will be discussed and shown in the next section.

### Diffs in In Operation

As mentioned in the previous section, there is an integer assigned for every byte of the shared object to keep track its time stamp value. If the value of the integer is zero, this means that the byte has never been changed. Thus, when a program requests the shared object handler for the latest diffs, the byte does not have to be sent for synchronization.

The shared object handler assigns an integer for every program that synchronizes with the shared object handler to keep track its time stamp. This time stamp value indicates the latest modifications received. For instance, if a program's time stamp



**Figure 1.6.** Diff Network Message

is two, it will only receive byte modifications that have a time stamped value of greater than two. This avoids unnecessary bytes to be sent during synchronization. After the latest modifications are sent, the shared object handler updates the time stamp value of the program to the present one.

The network message format on how the shared object handler relates which bytes are currently modified and which are not is the same with format in the out operation. After the program receives the network message and applies the modifications to the locally linked shared object, it then performs a twin operation. This prevents newly accepted modifications from the in operation to be mistaken as part of the current user modification.

## 1.6 Examples

This section first presents a simple implementation of the one-way bridge problem. Then, it presents another simple, but complete implementation of the jacobi. Finally, it presents encapsulating C++/Tuple-lock operations within a class and deploys it to parallel problems like producer-consumer and sleepy barber problem.

### 1.6.1 One-Way Bridge Problem

In the one-way bridge problem, there is a bridge that has only one lane. Thus, cars cannot move in opposing direction. When there is a car that is currently moving in the direction from left to right, a car that is going to move from right to left has to wait until the bridge is not occupied. While another car that moves in the same direction is allowed to continue.

In the implementation below, the program uses one integer to handle the critical section problem. If this integer is 0, this means that there is currently no car occupying the bridge. If the integer is positive, this means that there is currently one or more cars moving in the direction from left to right and vice versa if it is negative.

CLASS DEFINITION:

```
class bridges {
public:
    GM_TUPLE tuple;
private:
    bridges():tuple("int"){};
} bridge;
```

CAR (LEFT TO RIGHT): CRITICAL SECTION

```
//check if no car or right direction
GM.rd(&bridge.tuple, >=0 | +1);
... move from left to right ...
GM.rd(&bridge.tuple, | -1);
```

CAR (RIGHT TO LEFT): CRITICAL SECTION

```
//check if no car or left direction
GM.rd(&bridge.tuple, <=0 | -1);
... move from left to right ...
GM.rd(&bridge.tuple, | +1);
```

### 1.6.2 Jacobi

The Jacobi iteration is used for solving Laplace's equation. Given a matrix or grid, Laplace's equation is a partial differential equation that is used to approximate the steady-state solution for points in the interior. In Jacobi iteration, the value of each grid point is determined through taking the average of its four neighbors. This process is done until the difference between the old value and new value is within some acceptable difference EPSILON.

In the implementation below of Jacobi, each process holds a row of the matrix.

In order for it to solve for the average value of an element in the matrix, it needs to acquire the values of its neighbors. Thus, before the row is computed, the process does an in operation. After the whole row is computed, it does an out operation to flush out the modifications. At the same time, it waits for the other processes to finish computing and flushing out their modifications. Then, the process retrieves these modifications and computes the row again. To simplify the implementation, the program is iterated for a fixed number of times. The whole distributed program is started by running the jacobi manager program, jacobi.cpp.

### JACOBI HEADER (jacobi.h)

```
#include "gm.h"
#include "time.h"

#define NUM_ITER 100
#define SIZE 4

class jacobi {
public:
    int element[SIZE][SIZE];
    GM_TUPLE synch;

    jacobi():synch("int")
    {
        srand(time(NULL));
        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
                element[i][j]=1+(int)(10.0*rand()/(RAND_MAX+1.0));
    };

    void compute_row(int row)
    {
        int sum, divisor;
        int temp_row[SIZE];

        for (int i=0; i<SIZE; i++)
        {
            sum=0;
            divisor=4;

            // add the value of the left neighbor
            if (i!=0) sum+=element[row][i-1];
            else divisor--;
```

```

        // add the value of the right neighbor
        if (i!=SIZE-1) sum+=element[row][i+1];
        else divisor--;

        // add the value of the up neighbor
        if (row!=0) sum+=element[row-1][i];
        else divisor--;

        // add the value of the down neighbor
        if (row!=SIZE-1) sum+=element[row+1][i];
        else divisor--;

        temp_row[i]=sum/divisor;
    }

    for (int i=0; i<SIZE; i++)
        element[row][i]=temp_row[i];
}
};

```

### JACOBI MANAGER (jacobi.cpp)

```

#include "jacobi.h"

int main() {
    GM gm("jacobi");
    jacobi matrix;

    // attaching matrix to make it visible globally
    gm.attach ("matrix", &matrix, sizeof(matrix));

    // spawning workers to help on doing the tasks
    gm.spawn("/home/geoffrey/gm/examples/jacobi/worker", SIZE);

    // wait for all workers to finish an iteration
    for (int i=0; i<NUM_ITER; i++)
    {
        gm.in(&matrix.synch, SIZE);

        // display matrix
        for (int j=0; j<SIZE; j++)
        {
            for (int k=0; k<SIZE; k++)
                cout << matrix.element[j][k] << " ";
        }
    }
}

```

```

        cout << endl;
    }

    cout << endl;
}
}

```

### JACOBI WORKER (worker.cpp)

```

#include "jacobi.h"

int main() {
    GM gm("jacobi");
    jacobi matrix;

    // attaching matrix to make it visible globally
    gm.attach("matrix", &matrix, sizeof(matrix));

    // wait for all workers to finish an iteration
    for (int i=0; i<NUM_ITER; i++)
    {
        matrix.compute_row(gm.id()-1);
        gm.out(&matrix.synch, | +1 | SIZE);
        gm.in(&matrix.synch, | -1 | 0);
    }
}

```

### 1.6.3 FIFO Queue

Perhaps, one of the most common distributed structures that are used in processing is the bounded buffer FIFO queue. Such a queue is commonly used in handling limited resources. Below shows the implementation of the FIFO queue.

```

template <class Type>
class fifo_queue {
protected:
    Type element[MAX];
    GM_TUPLE head, tail;
    GM *gm;

public:
    fifo_queue(GM *g, const char *name):
    head("int,int,int"),tail("int,int,int") {
        g->attach (name, this,sizeof(*this)); gm = g;
        g->twin(this); // an explicit call to twinning}
}

```

```

void insert (Type var) {
    int Pindex;
    gm->rd (&head, <MAX | +1, ?Pindex | =(@+1)%MAX, -);
    element[Pindex] = var;
    gm->out (&tail, | +1, -, Pindex | =(@+1)%MAX);}

Type remove() {
    int CIndex;
    Type temp;
    gm->in (&tail, >0 | -1, ?Cindex | =(@+1)%MAX, -);
    temp = element[CIndex];
    gm->rd (&head, "| -1, -, Cindex | =(@+1)%MAX);
    return temp;}

bool empty() {
    int NotEmpty;
    gm->rd (&tail, ?NotEmpty, -, -);
    if (NotEmpty) return false;
    else return true; }
};

```

The @ symbol in the example above specifies the value of the field. The tuple-lock head guards insert and the tuple-lock tail guards remove. Both the tuple-locks contain three integer fields. For the tuple-lock head, the first field keeps track of the number of spaces available in the queue. The second field keeps track of the position where the process can insert its item. The last field keeps track of the remove index. This allows the removes to be done in a FIFO behavior even though they are executing in parallel. The other way around is true for the fields of the tuple-lock tail. To aide in understanding the above implementation, the tuple-lock operation rd and out inside the method insert is interpreted.

The above prototype of the FIFO queue allows the concurrent execution of inserts and removes. Although inserts and removes use the same queue, inserting and removing should not be prevented from running in parallel since they are two different operations targeted on different elements of the queue. Insertion can be done in parallel as long as there is still enough space to insert because sequential accesses on the tuple-lock ensures each gets a different element index. Deletion can be done in parallel as long as there are enough items to be removed. Although, inserts can be done in parallel, however, they are not allowed to flush out its value until the earlier inserts have flushed out. This is in respect to the FIFO behavior of the queue. The same is also true for removing. For insert, the process doesn't have to collect the modifications of the queue. Thus, the operation rd is used and thereby saving on the messages passed. For remove, the same is also true, except it is the other way around. This is the specific behavior of the queue that the tuple-

Operation	match   modify   match	Interpretation
Rd	!MAX   +1	Is there still space available? If there is, capture one space.
Rd	&PIndex   =(@+1)%MAX	Where is the space available? Place it in PIndex. Set the pointer to the next free space in the circular queue.
Rd	-	Does not matter.
Out	+1	Increment the number of elements available in the queue.
Out	-	Does not matter.
Out	Pindex   =(@+1)%MAX	Are the earlier producers finished? If so, flush out my modification and increment the counter to signal the next in line to flush out his.

**Table 1.3.** Explanation of Insert Method in FIFO

lock can be tailored to. Below shows brief examples of re-using this structure on the classical producer-consumer and sleepy-barber problem. In this example, the resource is an integer. On a more complex problem, the resource can be another object or a structure.

### Producer-Consumer

In the producer-consumer problem, the producer produces while the consumer consumes. There can be more than one producer and one consumer at a time. The producer cannot produce when the buffers are all full. On the other hand, the consumer cannot consume when the buffers are all empty. The behavior of the system below is similar to the queue handling explained before.

PRODUCER:

```
GM gm("producer-consumer");
fifo_queue<int> my_queue(&gm,"Queue");
int element = gm.id();
my_queue.insert (element);
```

CONSUMER:

```
GM gm("producer-consumer");
fifo_queue<int> my_queue(&gm,"Queue");
int element = my_queue.remove();
```

### Sleepy Barber

The sleepy-barber problem is a problem about coordinating the barber with the customer. The barber can only give haircut to one customer at a time. While the barber is busy, customers who want a haircut may wait at the waiting room. The number of chairs in the waiting room is fixed. If the waiting room is full, the customer comes back again later. If there is no customer, the barber goes to sleep. If the barber wakes up and finds out that there is still no customer, it goes back to sleep again [20]. Below shows an implementation of the sleepy-barber problem. Since the `fifo_queue` lacks the functionality for coordinating the barber with the customer, the class is extended. By inheriting the `fifo_queue`, the programmer does not have to start from nil.

EXTENDED FIFO QUEUE:

```
<template class Type>
class my_fifo_queue:public fifo_queue<Type>{
protected:
    GM_TUPLE coord;
public:
    my_fifo_queue(GM *g, const char *name):
        fifo_queue(g,name), coord("int"){
    void wait (int num) {gm.rd (&coord, num); }
    void next (int num) {gm.rd (&coord, |=num);}
};
```

BARBER:

```
GM gm("sleepy-barber"); my_fifo_queue<int> my_queue(&gm,"Queue");
while (1) {
    if (my_queue.empty()) /* sleep */;
    else my_queue.next(my_queue.remove());}
```

CUSTOMER:

```
GM gm("sleepy-barber"); my_fifo_queue<int> my_queue(&gm,"Queue");
my_queue.insert (gm.id()); my_queue.wait (gm.id());
```

## 1.7 Results

The system is tested on a cluster of computers running on Red Hat Linux version 6.1. The cluster is composed of seven computers equipped with dual PIII 500 MHZ symmetric multi-processor. It is connected through a 100Base-T fast Ethernet

switch. The cluster is lightly loaded and is not performing any major computational work. Beneath the system is PVM version 3.4. This is the latest PVM version as to the date of the experimentation. PVM is used by the system to transfer messages from one process to another. Each of the tests below is done 30,000 times. The average of the tests is then placed as the result.

### 1.7.1 Startup Costs

The startup costs of the system are divided into three kinds. First is the cost of the system initialization. System initialization includes spawning the central registrar and initializing it. This happens when a process creates a virtual distributed group. A virtual distributed group is created when no such group exists. This only happens to one process within a distributed group and it is the first process that tries to enroll into the group. Second is the cost of the process initialization. This includes enrolling itself into the distributed group. Third is the cost of the process attaching its local memory to the shared object. The first two costs happen when class GM is instantiated, as shown in the first line below. The third cost happen when the attach method is invoked, as shown in the second line of the example.

```
GM gm ("example");
gm.attach ("object", &object, sizeof(object));
```

Startup Costs	Time (msec)
System Initialization	1.653
Process Initialization	1.069
Attaching Cost	1.057

**Table 1.4.** Startup Costs

### 1.7.2 Tuple-Lock Costs

The tuple-lock operations in, out and rd are tested against different variables. In the table below, these operations are tested again increasing number of tuple-locks. Contrary to Linda, the results in table show that even as the number of tuples increases, the time needed to perform the tuple operations remain approximately the same. For the operation rd, it is around 0.16-0.17 milliseconds. This shows that the increasing number of tuples does not affect or have little bearing on the time needed to perform tuple operations. The results also show that the in and out operations perform significantly slower than the rd operation. For obvious reason, this is due to the larger messages sent and received as the modifications to the shared object are packed within. Thus, in times when the tuple-lock is used only

for coordination purposes, one can opt to use the rd operation rather than the in or out to be more efficient.

Number of Tuples	In (msec)	Out (msec)	Rd (msec)
1,000	6.443	9.191	0.164
2,000	6.437	9.174	0.171
4,000	6.445	9.195	0.160
8,000	6.438	9.181	0.167
16,000	6.447	9.198	0.171
32,000	6.455	9.183	0.170
64,000	6.441	9.163	0.172
128,000	6.440	9.169	0.162
256,000	6.440	9.202	0.167
512,000	6.439	9.194	0.170
1,024,000	6.445	9.190	0.173

**Table 1.5.** Tuple-lock Operations Cost vs. Increasing Number of Tuples

In the table below, the operations are tested against increasing sizes of the shared object. As the size of the shared object increases exponentially, the time needed to perform the in and out operation also increases exponentially. This is because the in and out operation also sends and retrieves the updates to the shared object. The out operation performs significantly slower than the in operation as the size of the shared object increases. This maybe due to the extra time incurred in performing, organizing, storing, compressing and garbage collecting the modifications or diffs to the shared object. The time needed to process diffs is proportional to the size of the object. As mentioned earlier, the rd operation only changes the tuple-lock contents and does not concern itself with the size of the object. Thus, the operation cost of rd is fairly constant.

Size of Object (bytes)	In (msec)	Out (msec)	Rd (msec)
1,000	1.694	1.772	0.167
2,000	3.207	3.803	0.177
4,000	6.449	9.193	0.169
8,000	12.305	24.603	0.175
16,000	24.379	74.735	0.173
32,000	48.230	245.431	0.172
64,000	96.690	916.235	0.177

**Table 1.6.** Tuple-lock Operations Cost vs. Increasing Sizes of Object

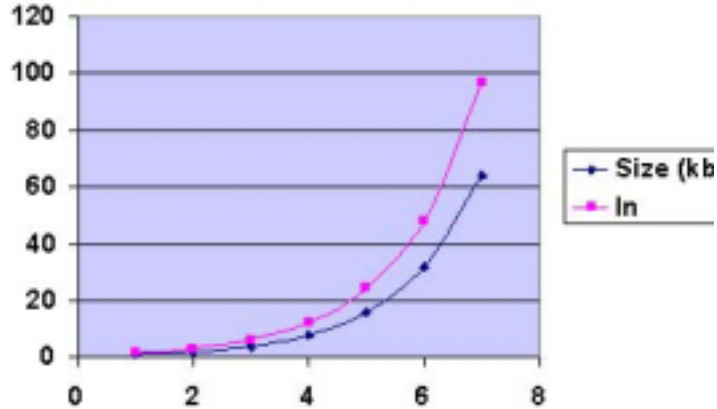


Figure 1.7. Time Spent: In Operation versus Size of Object

## 1.8 Advantages and Disadvantages

The tuple-lock design prevents false exclusion and false sharing [24]. False exclusion occurs when operations are excluded from each other when it need not be so. On the other hand, false sharing occurs when two objects reside in a page and the page was invalidated as a result of a modification to either one of the objects. Since there is clear definition of which tuple is guarding which object, a modification to one object only invalidates that object and would not affect the other. In addition, related objects can be grouped together and be guarded by one or more tuples. This allows pre-fetching of modifications when one of the objects is accessed.

Tuple-lock provides flexibility. Using the tuple-lock design, the programmer can implement the type of lock he needs at a particular moment. For instance, while a MRSW prohibits concurrent writes to an array, a tuple-lock with the appropriate locking can allow this and prevent false exclusion. Furthermore, tuple-locks can be tailored specifically to a type of data structure. This provides better efficiency. An example is tailoring tuple-locks to handle the bounded buffer FIFO queue. While one is inserting, another should not be prevented from removing an item even if they are using the same queue.

Since tuple-locks have fixed locations, inequality operations are more feasible than Linda. Inequality operations are more costly for Linda because such an operation satisfies more scopes and hashing may then not be much helpful in limiting the scope and reducing the search delay. On the other hand, the scope for tuple-locks is well defined. Thus, the queries are limited to the corresponding tuple.

Unlike Linda, it is inherent for unrelated tuple-locks to be distributed. Since tuple-locks have their corresponding fixed location, they can be assigned to different servers. Furthermore, since queries for unrelated tuple-locks do not concern with

each other, this allows tuple-locks to take advantage of distributed and concurrent processing of tuple-lock queries belonging to different objects.

One of the disadvantages of tuple-locks is its inability to dynamically create tuple-locks to associate to an object. However, in a distributed program, usually the behavior of the shared objects is defined beforehand. Thus, the number of tuple-locks that would be needed for synchronization and coordination are usually fixed. Furthermore, the system only limits the dynamic creation and association of tuple-locks to object. It does not limit the dynamic creation of distributed objects. Thus, in times when dynamic creation of tuple-locks is in dire need, it can also be done through encapsulating it within an object. However, the association of the encapsulated tuple-lock object to another object would then have to be done manually.

Perhaps, a major disadvantage of tuple locks lie in its complexity in writing one. Writing tuple locks can vary from easy to difficult. It largely depends on the kind of lock one is implementing. As shown in section 2, writing a mutual exclusion tuple-lock is very easy; however, it would be harder to write a MRSW lock. Furthermore, writing a tuple lock to handle a bounded buffer FIFO queue can proved to be even more complex.

## 1.9 Conclusion

Tuple-locks provide flexibility to the programmer in defining his own locking schemes. This is seen to contribute to the efficiency of a distributed program through preventing false sharing, false exclusion, pre-fetching and providing locks that are tailored to the data structures. Moreover, since tuple-locks have fixed location, the increasing number of tuples does not affect the efficiency of tuple-lock operations. However, writing tuple-locks may still prove to be challenging for complex mechanisms.

## 1.10 Bibliography

- [1] A. B. Hastings. Transactional Distributed Shared Memory. Thesis Summary, Carnegie Mellon University, Pennsylvania, 1992.
- [2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press, Massachusetts, 1994.
- [3] A. L. Cox, P. Keleher and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of 19th Annual International Symposium on Computer Architecture Conference*, pages 13–21, 1992.
- [4] A. L. Cox, S. Dwarkadas and W. Zwaenepoel. An Evaluation of Software-Based Release Consistent Protocols. *Parallel and Distributed Computing: Special Issues on DSM*, 29: 126–141, 1995

- [5] A. L. Cox, E. D. Lara, C. Hu and W. Zwaenepoel. A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory. In *Proceedings of the Fifth High Performance Computer Architecture Conference*, 1999.
- [6] A. Mohindra and U. Ramachandran. A Survey of Distributed Shared Memory in Loosely-coupled Systems, 1991.
- [7] Alan Burns and Geoff Davis. *Concurrent Programming*. Addison-Wesley Publishers Ltd, Great Britain, 1993
- [8] C. Amza, A. L. Cox, S. Dwarkadas and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the Third High Performance Computer Architecture Conference*, pages 261–271, 1997.
- [9] David Gelernter and David Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *1992 International Conference on Supercomputing*, pages 417–427, 1992.
- [10] J.Chua, C. C. Co, J. Jimenez, S. R. Simbulan and P. Chan. PVM-DSM: A Distributed Shared Memory System for the Parallel Virtual Machine. In *Proceedings of 2nd Symposium on Distributed Computing*, pages 52–65, 1999.
- [11] Linda Group. Department of Computer Science, Yale University. <http://www.cs.yale.edu/Linda/linda.html>
- [12] Mark Walmsley. *Multi-Threaded Programming in C++*. Springer-Verlag London, Great Britain, 2000.
- [13] Nicholas Carrier and David Gelernter (1988). Linda: Some Current Work. Department of Computer Science, Yale University, 1988.
- [14] Nicholas Carrier and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4): 444–458, 1989.
- [15] Peter Keleher. Lazy Release Consistency for Distributed Shared Memory. Ph. D. Thesis, Department of Computer Science, Rice University, 1995.
- [16] Rami Drucker and Ariel Frank. A C++/Linda Model for Distributed Objects. Department of Mathematics and Computer Science, Bar-Ilan University, Israel, 1996.
- [17] Robert Lafore. *Object-Oriented Programming in C++*, 3rd Edition. Sams Publishing, USA, 1999.
- [18] S. S. Fu and N. Tzeng. Aggressive Release Consistency for Software Distributed Shared Memory. In *17th International Conference Distributed Computing Systems*, 1997.

- [19] Seyed H. Roosta. *Parallel Processing and Algorithms: Theory and Computation*. Springer-Verlag New York, Inc., 2000.
- [20] Stephen J. Hartley. *The Operating Systems Programming: The SR Programming Language*. Oxford University Press, Inc., New York, 1995.
- [21] V. Lo and B. Nitzberg. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8): 52–59, 1991.
- [22] V. Milutinovic, J. Protic and M. Tomaevic (1996). Distributed Shared Memory: Concepts and Systems. *Parallel and Distributed Technology: Systems and Applications*, 4(2): 63–79, 1996.
- [23] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, Inc, Massachusetts, 1992.
- [24] Yuen Chung-Kwong. Annual Review of Scalable Computing, volume 3 (2000), pages 148–214.
- [25] Yuen Chung-Kwong, M. D. Feng, W. F. Wong and J. J. Yee. *Parallel List Systems: A Study of Languages and Architectures*. T. J. Press (Padstow) Ltd, Padstow, Cornwall, Great Britain, 1993.
- [26] Yuen Jien Soo and Yuen Chung-Kwong. *Balinda C++: A Parallel Programming System on Cluster of Workstations*. School of Computing, National University of Singapore, Singapore, 2000.
- [27] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the Second High Performance Computer Architecture Conference*, pages 26–37, 1996.