Thread Programming in SIMPL

Muhammed F. Mudawwar
Department of Computer Science
The American University in Cairo, Egypt
mudawwar@aucegypt.edu

Abstract

Advances in parallel architectures in recent years fueled the demand of a new class of programming languages that can simplify and promote parallel programming on a wide variety of parallel computers. This article introduces a new explicitly parallel programming language, called SIMPL. Three constructs are designed to support parallelism and to coordinate threads. The **thread** statement is used for forking threads. The **lock** and **wait** statements are used for synchronization. SIMPL is designed to support functional and data parallelism and to run on a wide variety of architectures. The simplicity and expressiveness of the parallel constructs is demonstrated. The implementation of these constructs is also discussed.

1. Introduction

Parallel programming languages differ greatly in their support for parallel programming. Some languages, such as High Performance Fortran [Koelbel 1994], are designed to support data or array parallelism. Other languages, such as Ada [Ada 1995], are designed to support task or functional parallelism. Declarative parallel languages, such as Id [Nikhil 1991], favor implicit parallelism rather than explicit parallelism. These languages are designed to avoid side effects as much as possible. Parallelism is easily detected and extracted by the compiler rather than being specified explicitly by the programmer. Other researchers have favored more complex approaches of extracting parallelism directly from sequential languages. Examples of parallelizing and optimizing compilers are PARAFRASE [Kuck 1984] and SUIF [Hall 1996]. The problem with a parallelizing compiler is that parallelism can be obscured in a sequential algorithm, and it is very difficult to extract it. An explicitly parallel language, on the other hand, encourages the programmer to write parallel algorithms. Furthermore, a compiler for an explicitly parallel language is easier to develop.

This article introduces a new programming language called SIMPL. SIMPL is a research-oriented explicitly parallel language designed to simplify and promote the programmability of parallel computers, especially shared-memory multiprocessors. This language uses the shared memory model of communication, and allows the programmer to exploit functional and data parallelism. SIMPL is an acronym for Simplified Imperative Modular Parallel Language. This language provides three simple constructs for starting threads and for synchronization. In addition, this language supports parameterized types, polymorphic functions, interfaces and modules. A complete definition of this language is provided in [Mudawwar 1998].

The next section discusses previous research on parallel languages. Section 3 presents three SIMPL constructs designed for parallel execution and synchronization. Examples on how to use these constructs in parallel programs are shown in Section 4. Section 5 discusses the issues related to the implementation of the parallel constructs.

2 Previous Work on Parallel Languages

This section focuses on parallel languages that support shared memory as a means of communication. These languages need constructs for starting and terminating threads, as well as for thread synchronization. Some languages also support parallel loops and allow the programmer to decompose and distribute arrays on processors.

2.1 Parallel Execution

Parallel execution differs greatly between programming languages. The **parbegin / parend** (or **cobegin/coend**) compound statement provides a parallel region, in which all enclosed statements can

potentially execute in parallel. Parallel and sequential regions can be nested inside each other. A thread can be forked to each statement in the parallel region. This is a simple and structured way to support parallelism in a programming language. However, the compiler should choose how many of the eligible statements should actually be executed concurrently, depending on the granularity of statements and the overhead of parallel execution.

Some languages, like Modula-3 [Nelson 1991], present a fairly low-level view of threads. A thread is started by a call to a *fork* procedure, which returns a thread identifier that can be used later for synchronization. *Fork* takes a procedure parameter that tells the thread what to execute. Sequential languages, like C or C++ do not provide parallel execution, but can use thread libraries, such as the *pthread* library under UNIX systems. Thread libraries are not simple to use but do support parallel execution.

Other languages like Ada [Ada 1995], use a high-level concept of a task subprogram to support parallel execution. When a task subprogram is called, a new thread is forked. The caller and the called threads progress in parallel. In SIMPL, a thread statement is used to support parallel execution. A thread statement is simpler and more flexible to use than a task or thread function as will be demonstrated.

2.2 Parallel Loops

Data-parallel languages include some sort of parallel iterative control structure. This is known as a parallel loop and is given different names such as **doall**, **pardo**, **forall**, and **doacross**. All the iterations of a parallel loop can be executed in parallel, but the statements of a given iteration are executed sequentially. Some researchers use **doacross** loops to mean a loop whose iterations are executed in a pipelined manner, rather than in parallel. A parallel loop is used essentially to expose data-parallelism, mainly array-parallelism, where the amount of parallelism exposed is equal to the size of arrays. In SIMPL, a thread statement with an iteration space is used to expose data-parallelism.

2.3 Thread Synchronization

In order for two threads running asynchronously to coordinate their activities, a parallel language must provide a means of synchronization. Synchronization can further be subdivided into access control and sequence control. Access control is needed in parallel languages that use shared-memory as a means of communication. Access control is to restrict the access of a shared data object such that one thread can manipulate it at a time. Sequence control, also known as condition synchronization, is to coordinate the execution of threads such that they progress in a correct order.

Many synchronization primitives have been suggested for access and sequence control. These include mutexes or locks, semaphores, condition variables, event counters, and barriers. For a coverage of these primitives, see [Finkel 1996]. Language constructs include the join statement, conditional critical regions, and monitors. The SIMPL language provides two constructs for thread synchronization. These are the **lock** and **wait** statements. These two statements are sufficient and can be used to implement synchronization primitives, such as semaphores, and monitored types.

2.4 Decomposing and Distributing Data Arrays

Some variations of Fortran 90, most notably High Performance Fortran [Koelbel 1994], allow the programmer to specify how to decompose and distribute the data arrays on different processors. The programmer does this in three stages:

1. Declares one or more *templates*, or virtual processor domains

2. Aligns each distributed data array to another array or one of the templates

3. Distributes the arrays or templates onto the actual processor ensemble

The decomposition and distribution of data arrays improve locality of reference and reduce communication. However, this is an extra work for the programmer who should be more focused on the parallel algorithms and shared data structures in a program. SIMPL does not allow the programmer to decompose and distribute data arrays. It relies more on compiler optimizations [Lim 1999] and hardware caches to improve locality of reference.

2.5 Declarative Languages

A declarative language avoids explicit parallelism in favor of implicit parallelism, extracted by the compiler, the run-time system, or even the hardware. Declarative languages support the single-assignment rule and permit a variable to be defined once. Since the variables in a declarative language cannot be updated, an imperative language programmer may view them as runtime constants. Designers of declarative languages strive to make their languages free from side effects to promote referential transparency and program reasoning. SISAL [Bohm 1991] and Id [Nikhil 1991] are examples of implicitly parallel declarative languages designed to address some of the application domains in which imperative parallel languages are used.

Compilers for declarative languages must perform several important optimizations to improve the efficiency (speed and memory usage) of the generated code. To support the single-assignment rule, a declarative program often uses more memory than a corresponding imperative program. For example, since an array cannot be updated, a new array must be allocated and defined. A compiler can perform important build-in-place and update-in-place analyses to optimize this, and experience has shown that the analyses are effective [Cann 1992].

3 SIMPL Constructs for Parallel Execution

SIMPL provides three constructs only for parallel execution and synchronization. These are the **thread** statement, the **lock** statement, and the **wait** statement. These statements are discussed next.

3.1 SIMPL threads

SIMPL uses a simplified, yet powerful thread statement for parallel execution. A thread statement specifies a thread name, an *optional iteration space*, and has a body that consists of a list of statements. The general form of a **thread** statement is shown below:

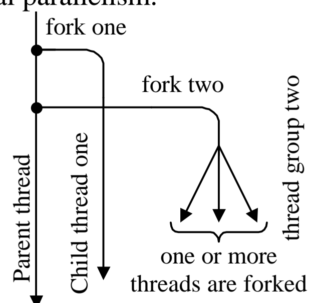
```
thread  name  iteration_space_opt  do  statement_list  end
```

There are two variations of the thread statement, depending on whether the iteration space is included or not:

1. A *simple thread statement* is one without an iteration space.
2. A *thread group statement* is one with an iteration space.

The order of statement execution in SIMPL is normally sequential. However, when a thread statement is encountered a new thread, called a *child thread* is forked. The forking thread is called the *parent thread*. The parent and child threads are scheduled independently and can potentially run in parallel. In the case of a thread group (with an iteration space), one or more threads are forked. An example of a simple thread and a thread group statement is shown in Program 1. In this example, up to *n* threads can be forked for thread group *two*, one for each iteration. When a small number of processors is available or when little work is done in each iteration of a thread group, a small number of threads are forked and the iteration space is distributed among these threads. Thus, a thread group in the SIMPL language is equivalent to a parallel loop in other languages. A thread group is designed to support data-parallelism, while a simple thread statement provides functional parallelism.

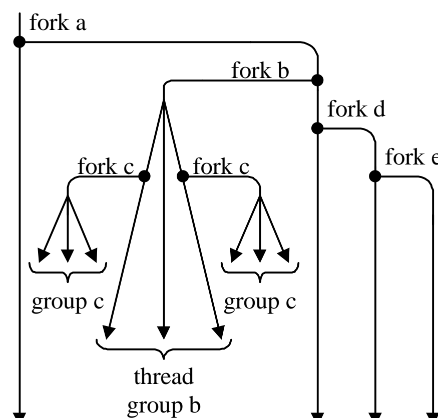
```
-- Simple thread statement
thread one do
   ...
end -- thread one
...
-- Thread group statement
thread two for i := 1 to n do
   ...
end -- thread two
```



Program 1: Forking threads and thread groups

Thread statements can be nested, exactly like other statements. A forked child thread can itself fork other threads and become a parent to them. Thus, a hierarchy of threads can result when thread statements are nested. Program 2 illustrates nested threads.

```
thread a do
  thread b for i := 1 to n do
    thread c for j := 1 to n do
      ...
    end -- thread c
    ...
  end -- thread b
  thread d do
    thread e do
      ...
    end -- thread e
    ...
  end -- thread d
  ...
end -- thread a
```



Program 2: Nested threads and their forking

A thread group is initially forked as a single thread. Depending on scheduling, number of processors, number of iterations, granularity of each iteration, and the current workload, a thread group can be split into a number of threads, each carrying a different subset of the iteration space. The runtime thread library handles the dynamic scheduling and splitting of a thread group. A programmer cannot specify this scheduling. A thread group with *n* iterations can be split into at most *n* threads, all sharing the same code, but each having a different iteration number. Therefore, the scheduling of a thread group is implementation dependent and may vary from one implementation to another.

It is important to notice that forking a thread group with *n* iterations is not identical to having a loop forking *n* threads as shown below. In the later case, the overhead of forking *n* threads can be substantial if *n* is large, especially if we were to execute these threads on a multiprocessor with a small number of processors.

```
-- Not the same as forking a thread group with n iterations
for i := 1 to n do
  thread x do
    ...
  end -- thread x
end -- for
```

3.2 The Lock Statement

When objects (variables) are shared by more than one thread, exclusive access to the shared objects must be guaranteed. The **lock** statement is used for this purpose. The lock statement specifies an object name to be locked and has a body that consists of a list of statements. The general form of a **lock** statement is shown below:

```
lock  obj_name  do  statement_list  end
```

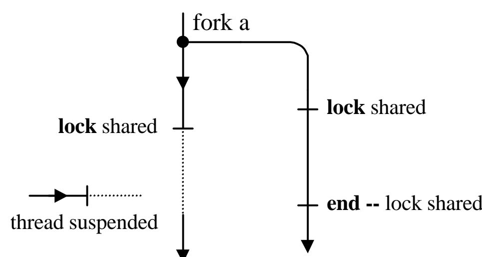
The compiler associates a *synchronization lock* with an object specified in a lock statement. When a thread executes a lock statement, it attempts to acquire the lock of the specified object name. If the lock is already acquired, the requesting thread is suspended. When a thread reaches the end of a lock statement, it releases the lock of the specified object. If other threads are waiting on this lock, one of them (the oldest waiting thread at the front of the lock's waiting queue) is allowed to resume execution.

Program 3 illustrates the use of a lock statement. A parent thread declares an object named *shared* and then forks thread *one*. Object *shared* can be accessed concurrently by thread *one* and its parent thread. To ensure exclusive access, the parent thread and its child thread *one* must use the **lock** statement.

An alternative approach to the lock statement is to define a *lock* type with two operations *acquire*() and *release*() for acquiring and releasing a lock object. This approach is, however, less structured than the lock statement and does not guarantee that the programmer will release a lock object at the end of a

critical section. The lock statement is more elegant because it can lock any object and guarantees its release at the end. Internally, the compiler takes care of associating a synchronization lock with each locked object. When a data structure is locked, the entire structure is locked and none of its elements can be accessed by other threads. A single lock is required in this case. However, when elements of a data structure are locked separately, an array of locks will be required. The lock statement can be used to implement a shared data structure, as will be demonstrated in the next section.

```
obj shared:sometype
thread one do
  lock shared do ... end -- lock
  ...
end -- thread one
...
lock shared do
  ...
end -- lock
```



Program 3: The lock statement and its effect

3.3 The Wait Statement

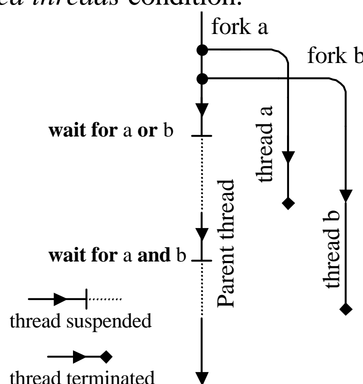
In addition to ensuring the exclusive access to shared objects, SIMPL allows threads to coordinate their execution. The **wait** statement is used for this purpose. There are two variations of this statement. These are the **wait for** and the **wait until** statements with the following syntax:

```
wait  for    terminated_threads
wait  until  boolean_expression
```

A parent thread sometimes needs to await the termination of one or more children threads before it can proceed. The **wait for** statement causes a parent thread to suspend execution until children threads, specified by name, terminate execution. This is a well-known form of synchronization called *join* synchronization. The *terminated threads* condition can further specify **and/or** synchronization. With **and** synchronization the parent thread awaits the termination of *all* the specified children threads. With **or** synchronization, the parent thread awaits the termination of *one* of the specified threads. Program 4 illustrates **and/or** synchronization. In the first case, the parent thread is suspended until either thread *a* or *b* terminates. In the second case, the parent thread is suspended until both threads *a* and *b* terminate.

We can also use the **wait for** statement to await the termination of a thread group. In this case, a parent thread is suspended until all the iterations (or threads) of a thread group terminate. Furthermore, we can mix **and** with **or** synchronization in the *terminated threads* condition.

```
thread a do
  ...
end -- thread a
thread b do
  ...
end -- thread b
...
-- wait for either thread
wait for a or b
...
-- wait for both threads
wait for a and b
```



Program 4: The wait statement with and/or synchronization

Since threads are specified as statements in a block, a thread name is visible only within its scope. This has two implications. First, the same name can be given to two different threads if they exist in different scopes, and second, a parent thread can see the names of its direct children threads, but not the names of its grandchildren threads. In the program shown below, the parent of thread *i* is awaiting the termination of its child thread *i*. It is not permissible for this parent thread to await the termination of threads *j* and *k* directly because they are not visible in its scope. Having the parent of thread *i* await the termination of thread *i* does not guarantee the termination of the children of thread *i*. However,

making thread *i* await the termination of its children threads will ensure the termination of thread *j* and *k* before the termination of thread *i* and its parent.

If a parent thread does not await the termination of one of its children threads then this child thread is said to be *detached*. If the main (root) thread of a program terminates and some of its detached children (or grandchildren) threads are still alive, the process and its threads are terminated by the underlying operating system because these threads operate in one address space.

```
thread i do
  thread j do ... end -- thread j
  thread k do ... end -- thread k
end -- thread i
wait for i
wait for j and k -- error (not visible)
```

```
thread i do
  thread j do ... end
  thread k do ... end
  wait for j and k
end -- thread i
wait for i
```

The **wait until** statement is used to support *condition synchronization*. This statement suspends the execution of a thread until the specified *boolean expression* (condition) becomes '*true*'. When a thread executes a **wait until** statement, it evaluates the boolean expression, and suspends itself if this expression evaluates to '*false*'. When a thread is suspended on a **wait until** statement, it becomes *sensitive* to all the variables specified in the boolean expression. When a second thread modifies any of the variables in the sensitivity list, it *triggers* the reevaluation of the boolean expression. When the boolean expression of a suspended thread evaluates to '*true*', it allows the suspended thread to resume execution.

An example of the **wait until** statement is shown in Program 5. In this example, a parent thread declares two objects *present* and *shared* that are shared by threads *producer* and *consumer*. The *producer* and *consumer* threads loop forever, producing and consuming a *shared* object on every iteration. The two threads synchronize their execution using the *present* variable that indicates the presence of a value in the *shared* object. Observe that enumeration literals, such as '*true*', are enclosed in single quotes to distinguish them from identifiers. Character literals, enclosed also in single quotes, are a special case of enumeration literals. String literals are enclosed in double quotes.

The SIMPL language does not impose any requirements on the run-time scheduling of threads and the exact timing of events. Therefore, when a thread is suspended on a **wait until** statement and a second thread modifies one of the variables specified in the sensitivity list of the first thread, it is wrong to assume that the boolean expression is reevaluated immediately. However, it will be reevaluated at a later time, depending on scheduling. If more than one modification occurs to the variables in the sensitivity list, the most recent modifications will be used when the boolean expression is reevaluated. Thus, care must be taken when using the **wait until** statement.

```
obj present: boolean := 'false'
obj shared: some_type

thread producer do
  while 'true' do
    obj local:some_type
    -- produce local object
    ...
    wait until not present
    shared := local
    present := 'true'
  end -- while
end -- thread producer
```

```
thread consumer do
  while 'true' do
    obj local:some_type
    wait until present
    local := shared
    present := 'false'
    -- consume local object
    ...
  end -- while
end -- thread consumer
```

Program 5: Producer-consumer synchronization using wait until

4 Programming with Thread, Lock, and Wait Statements

This section provides examples on the use of the **thread**, **lock**, and **wait** statements in the SIMPL language. The first example is the definition of a *semaphore* type, the second one is the definition of a *shared_queue* type, and the third example provides two parallel solutions to LU decomposition.

4.1 Generalized Semaphores

Traditionally, the *P* and *V* operations of a semaphore decrement and increment its value by 1. We can generalize these operations and specify a positive integer *n* to be subtracted or added to the internal value of a semaphore. A thread calling *P*(*n*) is blocked on a semaphore if it drives its value negative. A thread calling *V*(*n*) can wakeup zero, one, or more threads.

The interface and implementation of a generalized *semaphore* type is shown in Program 6. A type interface and a type implementation are normally separated and placed in different files. A type interface is exportable and contains the public members and function interfaces. A type implementation, which begins with the **private** keyword, is not exportable and contains the private members and the function implementations.

```
-- Type interface
type semaphore is
  function P(n:positive)
  function V(n:positive)
end -- type semaphore
-- Type implementation
private semaphore is
  obj value:integer := 0
  obj entry:boolean
  function P(n:positive) is
    lock entry do
      lock value do
        value := value - n
      end -- lock value
      wait until value >= 0
    end -- lock entry
  end -- P(n)
  function V(n:positive) is
    lock value do
      value := value + n
    end -- lock
  end -- V(n)
end -- private semaphore
```

Program 6: The interface and implementation of the type semaphore

The type *semaphore* has a private integer *value* member that stores the current value of the semaphore. The *P*(*n*) function decrements *value* by *n*, while *V*(*n*) increments it by *n*. The **lock** *value* statements in the *P* and *V* functions guarantee the correctness of the update. The **wait until** statement in the *P* function suspends a caller thread when *value* is negative. The **lock** *entry* statement in the *P* function blocks calling threads on the private member *entry* when a thread is currently blocked on the **wait until** statement. Thus, only one thread can be waiting on the **wait until** statement, while others wait on the *entry* lock. This has the effect of serializing the waiting of threads.

4.2 A Shared Queue Type

SIMPL does not support monitors or condition variables. Instead, the **lock** and **wait** statements can be used to implement them. A monitor is an abstract data type whose objects can be safely shared by many threads. The member objects of a monitored type are private and cannot be accessed directly, but rather indirectly through the public member functions. Several threads can call member functions concurrently but only one thread is allowed to be active within the monitor at a time. This is the classical definition of a monitor[Gehani 88], but in SIMPL we can do better and be less restrictive. We may allow more than one thread to be active within a monitored type. For example, Program 7 shows the implementation of a *shared_queue* type that can allow two threads to *enqueue*() and *dequeue*() concurrently. The *shared_queue* type is essentially a monitored type.

Function *enqueue* locks the *rear* index of a queue, while function *dequeue* locks the *front* index. If multiple threads call *enqueue* concurrently then they are serialized. The same thing occurs for function *dequeue*. However, it is possible to have two threads active in *enqueue* and *dequeue* concurrently.

The **wait until** statement replaces the need for *condition* variables, which are traditionally supported by monitors. Since the **wait until** statements are enclosed in **lock** statements in the *enqueue* and *dequeue* functions, only one thread can be waiting on them at a time. A thread calling *enqueue* is blocked on the **wait** statement if the queue is full (*count* = *n*). A thread calling *dequeue* is blocked on the **wait** statement if the queue is empty (*count* = 0). These threads are allowed to resume execution later when the queue is no longer full (for *enqueue*) or no longer empty (for *dequeue*). Observe that *count* is locked before updating it to ensure a correct update.

```
-- Interface of type shared_queue
type shared_queue{n:integer,t:type} is
  function enqueue(e:t)
  function dequeue():t
end -- type shared_queue

-- Type Implementation
private shared_queue{n:integer,t:type} is
  obj storage:array{n,t}
  obj front, rear, count:integer := 0

  function enqueue(e:t) is
    lock rear do
      wait until count < n
      storage[rear] := e
      rear := (rear+1) mod n
      lock count do
        count := count + 1
      end -- lock count
    end -- lock rear
  end -- enqueue()

  function dequeue():t is
    lock front do
      wait until count > 0
      result := storage[front]
      front := (front+1) mod n
      lock count do
        count := count - 1
      end -- lock count
    end -- lock front
  end -- dequeue()
end -- private shared_queue
```

Program 7: The interface and implementation of the type shared_queue

The type *shared_queue* is an example of a parameterized type with two parameters. The integer parameter *n* specifies the size of the private array *storage* and the type variable *t* specifies the *storage* element type. The actual values of *n* and *t* are specified when objects of this type are declared [Mudawwar 2000]

There is a similarity and a difference between the **lock** and **wait** statements. The similarity is that both statements are used to block threads. The difference is that if there are many threads blocked on the synchronization lock of a **lock** statement, only one of them is allowed to resume. However, if there are many threads blocked on the same **wait** statement, all of them are allowed to resume execution when the boolean condition is satisfied. If this is not desirable then the **wait** statement can be enclosed in a **lock** statement. This will ensure that only one thread can access the enclosed **wait** statement and block on it at a time.

4.3 LU Decomposition

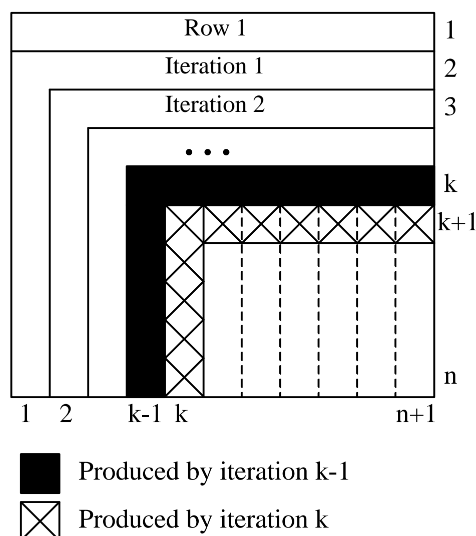
An example on thread groups is the well-known LU decomposition (gaussian elimination) algorithm. Two parallel algorithms are show in Program 8. Function *LUD* illustrates the use of an inner thread

group to exploit data parallelism. A **wait for** statement is used after this thread group to ensure the termination of all its threads, before moving to the next outer loop iteration. Therefore, all threads forked in this function terminate before returning from this function.

An alternative solution is presented in function *LUD2*. A *boolean* matrix *p* is associated with the *production* of the elements in matrix *a*. The element *p*[*i*,*j*] indicates that the final value of *a*[*i*,*j*] is computed and hence can be consumed. All the elements in the referenced matrix *p* should be initialized to '*false*' before function *LUD2* is called. The outer loop in function *LUD* is replaced with an outer thread group in function *LUD2*. Synchronization is done using **wait until** statements. Return from function *LUD2* occurs as soon as the thread group *outer* is forked. Thus, the caller thread and the function threads can progress in parallel. Consumer threads (not shown in Program 8) can consume matrix elements separately as soon as they are produced. Function *LUD2* is more aggressive at exploiting parallelism than function *LUD*, but requires an additional *boolean* matrix *p* for synchronization. Other variations are possible. In general, the programmer can control the parallelism to be exposed and the overhead of synchronization.

```
function LUD {n:integer}(obj a:matrix{n,n+1,real}) is
  for k := 1 to n-1 do
    thread inner for i := k+1 to n do
      a[i,k] := a[i,k]/a[k,k]
      for j := k+1 to n+1 do
        a[i,j] := a[i,j]-a[i,k]*a[k,j]
      end -- for
    end -- thread inner
    wait for inner
  end -- for
end – function
```

```
function LUD2{n:integer}
             (obj a:matrix{n,n+1,real},
              obj    p:matrix{n,n+1,boolean})
is
  thread row_1 for j := 1 to n+1 do
    p[1,j] := 'true'
  end -- thread row_1
  thread outer for k := 1 to n-1 do
    wait until p[k,k]
    for i:= k+1 to n do
      a[i,k] := a[i,k]/a[k,k]
      p[i,k] := 'true'
    end -- for
    thread inner for j := k+1 to n+1 do
      wait until p[k,j]
      for i := k+1 to n do
        a[i,j] := a[i,j] - a[i,k]*a[k,j]
      end -- for
      p[k+1,j] := 'true'
    end -- thread inner
  end -- thread outer
end -- function LUD2
```



Program 8: LU decomposition without pivoting.

The figure associated with function *LUD2* depicts the production of elements of matrix *a*. The strip covered in black is produced by iteration *k*-1 of the *outer* thread group, while the strip identified by crosses is produced by iteration *k*. The production of the subcolumn *a*[*k*+1 **to** *n*, *k*] is synchronized by the production of *a*[*k*, *k*], as indicated by *p*[*k*, *k*]. The update of element *a*[*i*, *j*], in the submatrix *a*[*k*+1 **to** *n*, *k*+1 **to** *n*+1] is synchronized by the production of element *a*[*k*, *j*]. The production of element *a*[*k*+1, *j*] is delayed until all the elements in the subcolumn *a*[*k*+1 **to** *n*, *j*] have been updated.

Functions *LUD* and *LUD2* are examples of polymorphic functions parameterized by the size *n* of matrix *a*. *n* is called a *hidden parameter* and is not passed explicitly when these functions are called. The hidden parameter *n* is inferred and passed implicitly for each function call [Mudawwar 2000]. *n* is considered a constant and cannot be modified in the bodies of functions *LUD* and *LUD2*. Matrices *a* and *p* are passed by reference.

5. Implementation Issues

This section discusses the issues related to the implementation of the **thread**, **lock**, and **wait** statements.

5.1 Implementing Threads

The connection between programming languages and operating systems is especially close in the area of parallel programming. Modern operating systems provide *threads* for supporting parallelism. Each thread has its private program counter, stack, and register context [Vahalia 1996]. Unlike UNIX processes that have separate address spaces, threads belong to and share the same address space. A SIMPL compiler can take advantage of existing thread libraries under UNIX systems. A thread group is forked initially as a single thread with an iteration space. Depending on the number of active threads and processors in the system, a thread group can split itself into two or more thread groups, each having part of the iteration space.

To allow SIMPL programs to run on a network of workstations, we have developed a virtual machine called VMTD (Virtual Machine for Thread Distribution). This is essentially a thread library that can distribute and migrate threads on a homogeneous network of workstations. A pool of workstations is initially selected and a process is forked on each workstation. We chose a homogeneous network of workstations to run the same binary code on all workstations and to simplify the implementation of a shared data segment. The shared data segment is actually replicated on all workstations. Consistency is maintained by broadcasting changes in the shared segment. Because the network is homogeneous and the same binary code is used, we were able to use the same virtual address (pointer) to access the shared data segment in all processes. Another feature of VMTD is that it simplifies the migration of threads. When a thread is created, a thread identifier is added to the local scheduling queue, but no thread context is created and the thread is not committed to its current workstation. When a workstation becomes idle, load balancing takes place and uncommitted threads are migrated. Only thread identifiers are transferred, because uncommitted threads have no context and the code is already replicated on all workstations. When a thread is scheduled to run on a workstation, a new context (stack and private area) is allocated and the thread becomes committed. Once committed to a workstation, a thread cannot migrate. A committed thread continues to run until it terminates or it suspends on a synchronization statement. Scheduling of threads within a process is non-preemptive, while the scheduling of processes within the kernel is preemptive.

5.2 Implementing the Lock Statement

As mentioned earlier, the compiler associates a synchronization lock with each locked object. A lock is a boolean flag and a waiting queue of threads. A lock is acquired at the beginning of the lock statement and is released at the end. When a lock is currently acquired, a thread suspends itself of the waiting queue until the lock is released by another thread.

Locks are implemented using primitive hardware instructions, such as *test-and-set* or *swap*. They are also implemented as part of existing thread libraries. Thus, lock statements can be implemented quite easily. One possible variation is to insert suspended threads at the end of the scheduling queue rather than putting them on the waiting queue of the acquired lock. By the time they run again and reacquire the lock, the lock might have been released. This will simplify the implementation of a lock to a boolean flag. Waiting queues are no longer necessary.

5.3 Implementing the Wait Statement

The wait statement with its two variations, **wait for** and **wait until** can be implemented in one consistent way. A boolean variable is associated with the termination of each thread and thread group that is awaited in a **wait for** statement. Thus, the thread termination condition can be formulated as a boolean expression and a **wait for** statement can be reduced to a **wait until** statement.

A wait statement is implemented as a while loop. As long as the boolean expression is not satisfied, a waiting thread is put at the end of the scheduling queue. When the same thread runs again, it will reevaluate the boolean expression. The thread will be able to resume execution when the boolean expression is satisfied. Observe that this is not equivalent to busy waiting.

```
wait  until  boolean_expression
```

is translated as:

```
while not boolean_expression do
  put this thread at the end of scheduling queue
end
```

Putting a waiting thread at the end of the scheduling queue is less costly and more efficient than implementing a waiting queue. By the time, a thread gets its turn to run again, it is likely that the condition it is waiting on becomes satisfied.

6 Conclusion

This paper has demonstrated that thread programming can be simplified if language constructs are designed properly. The SIMPL **thread**, **lock** and **wait** statements are used to solve a wide variety of problems. These constructs are simple enough and can be implemented quite easily. We have designed and implemented a virtual machine, called VMTD, to distribute and migrate threads on a homogeneous network of workstations. Future research in this direction is to exploit hardware support for the efficient implementation of these constructs.

References

[Ada 1995] *Ada 95 Reference Manual*, ISO/IEC 8652:1995, International Organization for Standardization/International Electrotechnical Committee, January 1995.

[Bohm 1991] Bohm A. P., Oldehoeft R. R., Cann D. C., and Feo J. T., *Sisal 2.0 Reference Manual*, Technical Report CS-91-118, Computer Science Department, Colorado State University, Nov. 1991.

[Cann 1992] Cann D. Retire Fortran? A Debate Rekindled, *Communications of the ACM*, August 1992, p. 81-89.

[Finkel 1996] Finkel R. A., *Advanced Programming Language Design*, Addison-Wesley Publishing Company, 1996.

[Gehani 1988] Gehani N. and McGettrick A. D. editors, *Concurrent Programming*, Addison-Wesley, 1988.

[Hall 1996] Hall M., Anderson J., Amarasinghe S., Murphy B., Liao S., Bugnion E., and Lam S., Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer*, December 1996, (special issue on multiprocessors)

[Koelbel 1994] Koelbel C., Loveman D., Steele G., and Zosel M., *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[Kuck 1994] Kuck D., Kuhn R., Leasure B. and Wolfe M., The Structure of an Advanced Retargetable Vectorizer", in *Tutorial of Supercomputers*, IEEE Press, 1984, p. 163-178.

[Lim 1999] Lim A. W., Cheong G. I., and Lam M. S., An affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication, in *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, June 1999.

[Mudawwar 2000] Mudawwar M., Parameterized Types and Polymorphic Functions in SIMPL, in *Proceedings of the 8th International Conference on AI Applications,* Feb. 2000.

[Mudawwar 1998] Mudawwar M., *SIMPL: Language Definition*, Technical Report, Computer Science Department, the American University in Cairo, June 1998.

[Nelson 1991] Nelson G., *Systems Programming with Modula-3*, Prentice Hall, 1991.

[Nikhil 1991], Nikhil R. S., *Id (Version 90.1) Reference Manual*, Technical Report CSG memo 284-2, MIT Lab for Computer Science, Cambrigde, MA, July 1991.

[Vahalia 1996] Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, 1996.

[Wolfe 1996] Wolfe M., High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.