Parallel Alchemist: a Visual Environment for Parallel Software Development with Shared Memory Programming Models

Márcio de Oliveira Barros*
cuco@nce.ufrj.br

Júlio Salek Aude**
salek@nce.ufrj.br

*** NCE/UFRJ**

**** IM/UFRJ and NCE/UFRJ**

Núcleo de Computação Eletrônica
Federal University of Rio de Janeiro
Cidade Universitária, RJ, Brasil
CEP: 20001-970 P.O. Box: 2324

Abstract

This paper presents a visual programming environment for parallel software development. Parallel Alchemist is based on multithreaded programming and uses shared memory for communication. The software development process is almost visual, allowing users to build complex programs based on primitive programming blocks. Parallel Alchemist works upon meta-schemes of parallel programming models and generates code in any model the user describes with these meta-schemes, including the Mulplix native programming model, under development for the Multiplus distributed shared memory multiprocessor. Parallel Alchemist is implemented in Java and is currently available on Solaris platform. Examples of meta-schemes and generated code are presented and discussed.

Resumo

Este artigo apresenta um ambiente de desenvolvimento visual de software paralelo. Alquimista se baseia em multithreading e utiliza memória compartilhada como o meio de comunicação entre as threads. O processo de desenvolvimento de software é integralmente visual, permitindo a construção de programas complexos a partir de blocos de programação primitivos. Alquimista utiliza meta-esquemas de modelos de programação paralela e gera código em qualquer modelo descrito por um meta-esquema, inclusive o modelo nativo de programação paralela Mulplix, em desenvolvimento para o multiprocessador Multiplus. Alquimista está implementado em Java e disponível para a plataforma Solaris. Exemplos de meta-esquemas e código gerado são apresentados e discutidos neste artigo.

Keywords: software development tools, parallel computing, multithreading

1. Introduction

Software development, as stated by Barry Boehm [BOE76] in the late 1960's, is a difficult task. In the last three decades, this scenario has not changed. As programming languages and environments evolve, software complexity evolves even faster. The need for distributed and parallel computation increases such complexity considerably.

On the other hand, visual programming is a potentially powerful and appealing technique for building complex applications. As a consequence, an increasing number of research works on visual programming languages [HIL92, REP95] and environments [KAR95, ING88] has been reported lately.

Nevertheless, the application of visual programming techniques to the development of parallel programs is an area that needs further development. A few efforts on the use of these techniques have been reported. P-Rio [CAR95] and VPE [NEW95] are visual environments designed for the generation of PVM-based code for parallel systems. The Linda Program Builder [AHM94] is a user-friendly programming environment that allows the construction of parallel programs from frameworks and code templates. Code [BRO95] is a visual environment that allows the composition of a parallel program from sequential ones. Code can produce parallel programs for shared-memory and distributed-memory architectures, using PVM, MPI and Pthreads. Enterprise [SCA93] is a programming environment that allows the visual composition of parallel programs from sequential programs. Enterprise uses the message-passing paradigm, handling communication and synchronization between the sequential parts of the parallel program.

This work describes the development of **Parallel Alchemist**, a visual environment that generates C code for shared memory based parallel programming models. Parallel Alchemist helps software developers with little expertise on parallel programming. Alchemist's features, however, are also useful for experienced parallel programmers, who need code portability across different parallel programming models.

The environment is an evolution and a generalization of a previous development effort, Parallel Wizard, which is a parallel software development environment, designed only for the Mulplix parallel programming model [AZE93]. Parallel Alchemist, however, is able to cope with different shared memory based programming models.

The environment is based on multithreading. Threads are lines of control flow that can be executed in parallel on different processors. Every program has, at least, one thread, known as the main thread, which is responsible for the creation of other threads. Shared memory is used for thread communication.

Parallel Alchemist defines a parallel programming meta-model. This meta-model defines the parallel entities and activities used during the software development process. The environment uses meta-schemes of existing parallel programming models to translate the meta-model representation to a particular model application interface. Parallel Alchemist can generate code for any model represented by meta-schemes. Currently three models are implemented: Solaris threads [GRA95], Posix threads (Pthreads) [IEE94] and the Mulplix native model, designed to be used within the distributed shared memory Multiplus multiprocessor [AUD96].

Section 2 presents the Alchemist's development process, describing how software is built within the environment. Section 3 presents the basic programming blocks, which are primitive visual constructors used to define the software behavior. Section 4 describes the parallel meta-model used by the environment. Section 4.1 presents a brief description of the Mulplix native programming model. Section 5 presents meta-schemes already generated within Alchemist. Section 6 demonstrates how code is generated by the environment. Section 7 presents two programming examples. Section 8 presents the Alchemist's current implementation. Conclusion and directions for future work are presented in Section 9.

2. The Development Process

Parallel Alchemist is based on the structured software development paradigm [PRE92] and uses the **C** programming language [KER88]. The structured paradigm states the functional decomposition of a complex problem into smaller, much simpler, problems. The program execution flow is decomposed into several routines and controlled by a master routine, where execution starts and ends.

The C language implements the structured paradigm. The environment uses this language for code generation, due to its wide acceptance among developers and its availability on many operating systems.

On Parallel Alchemist, each parallel program consists of a set of shared variables, representing the shared memory, and a set of modules, representing the functional part of the system. Each module consists of a set of routines. Figure 1 presents the hierarchical representation of a program.

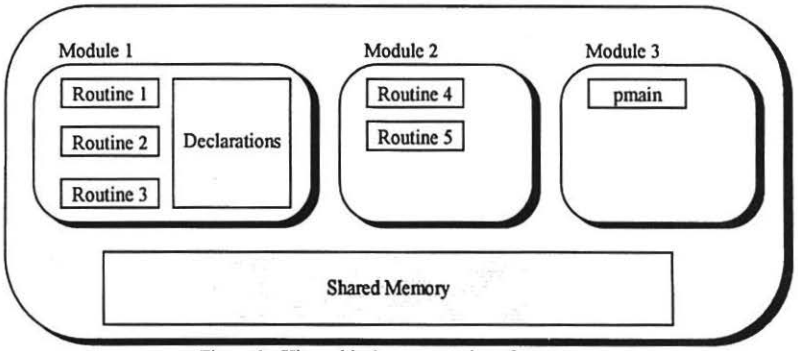

Figure 1 - Hierarchical representation of a program

Each shared variable is treated as an atomic element, described by its name and type. This type must conform to any type defined by the C language, declared in libraries used by the program or declared by the user. The type of a shared variable consists of a type prefix and a type suffix.

The type prefix is the primary type of the variable, such as integer types, single or double precision floating point, structures, unions, enumerations or pointers. The type suffix is used in array declarations, indicating the number of elements in the arrays.

Each module consists of a declaration section and a set of routines. Within the declaration section, the user can define types, create macros and constants, include header files and insert conditional compilation code, using C language statements.

Each routine is described by a behavioral graph, a directional graph that expresses the semantics of the routine using visual constructors. The nodes of a behavioral graph are called basic programming blocks. The connections between nodes represent possible execution flows inside the routine.

The program execution flow starts and ends in a routine called **pmain**, that receives two parameters, *argc* and *argv*, as the main function of the C language. The parameters indicate, respectively, the number of arguments within the program command line and a list of strings containing these arguments.

3. The Programming Blocks

Behavioral graphs are the central part of the software development process within Parallel Alchemist. Each behavioral graph describes a routine. The graph composition process is almost visual: the user selects the desired programming block, indicates where it must be inserted into the routine and the environment automatically creates connections from existing blocks to the newly created one.

Every programming block contains a set of properties. These properties are used during code generation. They describe details about the semantics of each specific programming block.

Every behavioral graph contains, at least, one programming block: the function declaration block. This block is created automatically by the environment, when a new routine is declared. It has a single property, called declarations, where the user must define the routine prototype and local variable declarations, using C code.

There are four types of common activity blocks. They define the one-dimensional flow of control, that is, the flow of execution inside a single thread.

Statement blocks allow the user to specify commands to be executed when that programming block is reached. These commands may include mathematical computation, file system I/O and calls to the C library routines. This programming block has a single property, called commands, in which the C statements to be executed are specified.

Condition blocks create a pair of branches on the behavioral graph, associated with the value of a condition, specified by the single property of this programming block.

Loop blocks create a single branch on the behavioral graph. This branch is repeatedly executed, until an associated condition evaluates as false. The properties of this programming block define a statement to be executed before the loop, a statement to be executed after each loop step, the condition associated with the loop and the moment when the condition is evaluated (after or before each loop step).

Routine call blocks specify the execution of a routine defined by other behavioral graph. These nodes allow the environment to determine which routines are executed in parallel. The properties of this programming block specify the routine to be executed, its parameters and the recipient of the routine resulting value.

There are six types of parallel activity blocks. They define the life cycle of the threads, indicating when they are created or destroyed.

The thread spawn block allows the creation of a single thread executing a routine, specified by another behavioral graph. The properties of this programming block are the thread routine, the parameter to be passed to the thread, a recipient to the thread identification and the kind of thread to be created: detached or non-detached. The differences between these kinds of threads will be presented in the next section.

The synchronous thread spawn block allows the creation of a group of threads, all executing the same routine, specified by another behavioral graph. The current thread will suspend its execution and wait for the conclusion of the new threads. The properties of this programming block specify the thread routine and the number of threads to be created.

The thread identification block allows a thread to know its own identifier. Each thread has a unique numerical identifier. This block has a single property, indicating a recipient for the thread identifier.

The thread wait block suspends the execution of the current thread until the conclusion of a specific non-detached thread. The single property of this programming block specifies the desired thread, through its identifier.

The thread kill block cancels a running thread. The single property of this programming block is the identifier of the thread to be canceled.

The thread termination block indicates the termination of the current thread. It has no properties. If this block is reached in a routine not associated with a thread, the whole program execution is concluded.

4. Alchemist's Parallel Meta-Model

The parallel programming blocks described in Section 3 define part of the Alchemist's parallel meta-model, an abstract programming model used to generate software code for any particular parallel model.

We call a parallel programming model a set of routines that act as interface between an application program and the parallel resources of an operating system. Examples of parallel programming models are POSIX Threads (Pthreads), Parallel Virtual Machine (PVM) [GEI94], Message Passing Interface (MPI) [MPI95], Mulplix native parallel system calls, Solaris Light Weight Processes (LWP) [SUN95] and Solaris Threads.

The currently available parallel programming models can be classified into two groups: shared memory based models and message-passing based models. Shared memory based models use a global memory to keep information accessible to every thread of a program. Pthreads, Mulplix and Solaris Threads are examples of shared memory based models.

Message-passing based models use information channels to send messages between threads. Each thread has buffers, where any message is kept until the thread attempts to receive it. PVM and MPI are examples of message-passing based parallel models.

Parallel Alchemist uses multithreading and shared memory. The environment does not assume a specific parallel programming model. It works upon meta-schemes of parallel programming models and generates code for any model described by a meta-scheme.

The Alchemist's meta-model describes what functionality a particular parallel programming model must have and the developer's interface to that functionality. The meta-model considers two related entities - threads and mutual exclusion semaphores - and activities related to these entities.

Threads are the basic source for parallel processing. A thread runs a particular routine in parallel with other threads of the program. Each thread has a unique numerical identifier, which is used in activities upon the thread. These activities refer to the creation of new threads, waiting for a thread conclusion, cancellation of a running thread and getting the thread identifier.

There are two kinds of threads: detached and non-detached. When a detached thread is created, no other thread can wait for its conclusion. The only possible control upon this thread is cancellation. When a non-detached thread is created, another thread can wait for its conclusion. This is useful for program synchronization.

Mutual exclusion semaphores, also called mutexes, are used to prevent multiple accesses to critical regions of code, that is, segments of code where shared resources can be accessed in parallel by several threads. The activities related to a semaphore are its declaration, creation, destruction, locking and unlocking.

4.1.The Mulplix Parallel Programming Model

The Mulplix native parallel programming model is part of the Mulplix operating system, a UNIX-like system designed to support medium-grain parallelism and to provide an efficient environment for running parallel applications within the Multiplus multiprocessor [AUD96]. Currently, we also have a Mulplix implementation under the Solaris platform.

The Mulplix native programming model is implemented at kernel level and accessed as system calls. These system calls are divided into three classes: thread manipulation, mutual exclusion semaphores and partial order semaphores.

The *thr_spawn* system call is provided for the creation of a group of threads. Mulplix threads are always created in groups and are always detached. To allow compatibility with the Alchemist's parallel meta-model, external routines have been implemented to create non-detached threads.

The *thr_spawns* system call allows the creation of threads in synchronous mode. If the thread creation is synchronous, the parent thread will suspend its execution until execution completion by all the children threads it has started.

Three additional system calls for thread control have also been made available within Mulplix. The first one, *thr_id*, returns the identification number of the current thread. Mulplix threads do not publish their identifications: only the thread itself knows its own numerical identifier. Again, to allow compatibility, external routines have been implemented to create threads and wait for them to return their identifiers. A similar solution was used for the implementation of the Pthreads model on top of Mulplix [BAR96].

The second system call, *thr_kill,* allows any thread to kill another thread within the same process. All the descendants of the killed thread are also killed. The last system call, *thr_term,* allows a forced termination of the current thread.

For the manipulation of mutual exclusion semaphores, system calls are provided for creating (*mx_create*), locking (*mx_lock*), extinguishing (*mx_delete*) and unlocking (*mx_free*) a semaphore. In addition, the primitive *mx_test* allows a thread to allocate a semaphore if it is free without causing the thread to wait if the semaphore is still occupied.

For partial ordering semaphores, which implement barrier-type synchronization, system calls for creating (*ev_create*), asynchronous signaling (*ev_signal*), waiting on the event occurrence (*ev_wait*), synchronous signaling (*ev_swait*) and extinguishing (*ev_delete*) an event are provided. The primitives *ev_set* and *ev_unset* have also been implemented to allow unconditional setting and resetting of an event. This may be useful in test, debugging or in error recovery procedures.

5. Alchemist's Meta-Schemes

Parallel Alchemist generates code using meta-schemes of parallel programming models. These schemes determine how the parallel meta-model maps to a particular parallel programming model. The environment can generate code for any parallel programming model described by a meta-scheme. Figure 2 presents Parallel Alchemist's code generation structure.

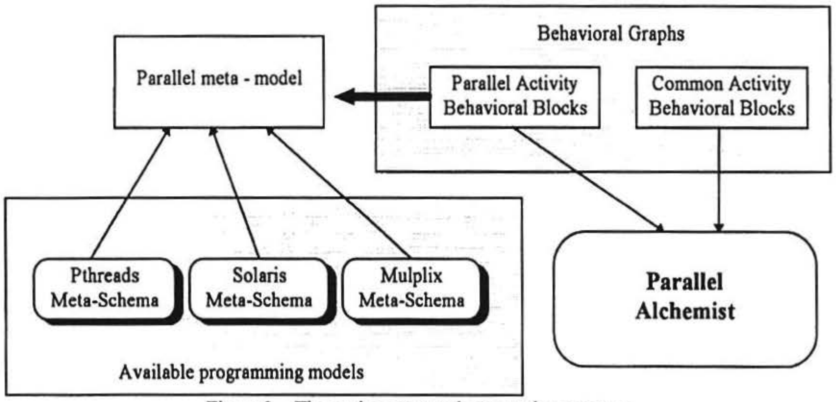

Figure 2 - The environment code generation structure

Each meta-scheme is a text file divided into four sections: headers, objects, mutex definitions and thread definitions.

The header section is a single line indicating the header files (i.e., the *.H* files) to be included in the modules using parallel activities, when code is generated using this parallel programming model.

The object section is another single line indicating the object files (i.e., the *.O* files) and the libraries to be linked with the modules generated by the environment. Sometimes, the parallel programming model does not provide the same interface required by the Alchemist's meta-model. To solve this problem, external conversion routines must be written to bridge the way between the two interfaces. The object files containing these routines must be indicated on the object files section of the model meta-scheme.

The mutex and thread definition sections consists of code patterns. These code patterns are parameterized by variables, which are filled by the environment, when generating code for an application. Variables are represented by an identifier, starting with a dollar sign.

The mutex definition section consists of five code patterns: declaration, initialization, destruction, locking and unlocking. Each code segment may use a variable, called Name, which specifies the mutex to be declared, initialized, destroyed, locked or unlocked.

The thread definition section consists of six code patterns: single thread creation, synchronous thread creation, thread waiting, thread termination, thread cancellation and thread identification.

The single thread creation pattern can use four parameter variables: the routine that will be executed within the thread, the argument to be passed to the thread, a recipient to its identifier and a flag indicating if the thread is detached.

The synchronous thread creation pattern can use two parameter variables: the routine that will be executed in association with the threads and the number of threads to be created. When synchronous threads are created, the current thread waits until their termination. The routine executed on a synchronous thread receives an integer argument, indicating its order on the spawned group.

The thread wait pattern uses a single parameter variable, the identifier of the thread whose conclusion will be waited for. This thread must be non-detached.

The thread cancellation pattern uses a single parameter variable, the identifier of the thread to be canceled. The thread termination pattern must contain code to conclude the execution of the current thread and receives no parameter variable.

The thread identification pattern must contain code to return the identification of the current thread. It receives the recipient for the identification, which may be a global or local variable, declared with an integer type. Table 1 presents the meta-schemes for the Pthreads and Mulplix parallel programming models.

Pthreads Model	**Mulplix Model**
HEADERS pthreads.h OBJECTS -lpthreads pbridge.o	HEADERS threads.h OBJECTS mbridge.o
MUTEX	MUTEX
Declaration pthread_mutex_t $Name;	Declaration MUTEX $Name;

Init
if (pthread_mutex_init (&$Name, NULL) < 0)
printf ("Init error: $Name\n");
	Init
	if (($Name = mx_create (1)) < 0)
	printf ("Init error: $Name\n");

Done
if (pthread_mutex_destroy ($Name) < 0)
printf ("Destroy error: $Name\n");
	Done
	if (mx_delete ($Name) < 0)
	printf ("Destroy error: $Name\n");

Lock if (pthread_mutex_lock ($Name) < 0)	Lock if (mx_lock ($Name) < 0)

printf ("Lock error: $Name\n");	printf ("Lock error: $Name\n");

Unlock
if (pthread_mutex_unlock ($Name) < 0)
printf ("Unlock error: $Name\n");
	Unlock
	if (mx_free ($Name) < 0)
	printf ("Unlock error: $Name\n");

THREADS	THREADS

Spawn
if (thr_create (&$Tid, $Function, $Arg) < 0)
printf ("Thread creation error: $Function\n");
if ($Detached)
pthread_detach ($Tid);
	Spawn
	if (($Tid = spawn ($Function, $Arg, $Detached)) < 0)
	printf ("Thread creation error: $Function\n");

SyncSpawn
if (sync_create ($Number, $Function) < 0)
printf ("Synchronous creation error: $Function\n");
	SyncSpawn
	if (thr_spawns ($Number, $Function, 0, 0) < 0)
	printf ("Synchronous creation error: $Function\n");

Wait
if (pthread_join ($Tid, NULL) < 0)
printf ("Wait error: %d\n", $Tid);
	Wait
	if (thr_wait ($Tid) < 0)
	printf ("Wait error: %d\n", $Tid);

Therm pthread_exit (NULL);	Therm thr_term ();

Kill
if (pthread_kill ($Tid, SIG_KILL) < 0)
printf ("Cancellation error: %d\n", $Tid);
	Kill
	if (thr_kill ($Tid) < 0)
	printf ("Cancellation error: %d\n", $Tid);

ThrID $Tid = pthread_self ();	ThrID $Tid = thr_id();

Table 1 - Meta-schemes for Pthreads and Mulplix parallel programming models

The Pthreads model does not directly allows synchronous thread creation. In the above example, the ***sync_create*** routine creates compatibility between this model and the parallel meta-model. This routine creates the required threads and waits for their conclusion, simulating a synchronous creation. The external routines required by the Mulplix model were discussed on section 4.1.

6. Code Generation

The code generation process occurs in the following steps. First, the user selects a parallel programming model for code generation.

Next, the environment discovers which routines can be executed in parallel. Routines that are executed only when the main thread is running do not need to be protected against multiple accesses to shared memory. To complete this step, the code generator follows the program execution flow. It marks routines called after a thread creation or executed within threads. These routines must protect theirs accesses to shared memory using mutual exclusion semaphores.

After determining which routines need to be protected, the code generator visits every program module, generating code for its routines and declarations. For each module, two code files are generated: the header file and the C code file.

The header file contains the declarations of the module. A reference to this file is included in every module generated by the environment. In addition, every module includes the header files of the selected parallel programming model. These files are defined in the parallel model meta-scheme.

The C code file contains code generated for the module routines. For each routine, its behavioral graph is followed and code is generated for each basic programming block.

The prototype and local variables of the function declaration block are copied to the destination file. The same happens to the commands specified in the statement block. Nested sequences of programming blocks, like loops and conditions, are replaced by C code structures that replicate their behavior. Routine calls are created based on the parameters and return recipients, specified during the behavioral graph construction.

Code generation for the parallel activity blocks is based on the parallel programming model selected by the user. The parameter variables in the code patterns for each parallel activity are substituted by theirs values, specified during the behavioral graph construction.

Shared variables can be declared as common variables or write-once variables. Accesses to common variables along behavioral graphs are protected by mutual exclusion semaphores. On write-once variables, the "write" operation is supposed to occur on a multithread-safe segment of code, that is, when only the main thread is running. As long as their values are constant, write-once variables need no protection against parallel accesses.

Every common shared variable is associated with a global mutex that protects the code segments where the variable is accessed. During code generation for a routine that can be executed in parallel, every programming block is checked against the use of each common shared variable.

If a programming block accesses a shared variable, a mutex locking code is generated before the code of the programming block itself. If the next programming block does not access the same shared memory, a mutex unlocking code is generated, after the code of the first programming block. Otherwise, the mutex unlocking is generated after the last block in sequence that accesses the shared variable. The locking and unlocking code depends on the parallel model meta-scheme.

Parallel Alchemist handles a list of multithreading safe functions (MT-Safe), that is, a list of C functions which are protected from parallel execution. The C standard library I/O functions, like *printf* and *scanf*, are examples of multithreading safe routines. They have internal protection against parallel execution of some critical regions of code.

If a common shared variable is accessed in parallel on a call for a MT-Safe function, the environment does not generate mutex lock and unlock code to protect the variable, since the called routine itself is already protected.

The mutual exclusion detection algorithm cannot detect indirect accesses to shared memory. Since C is a very generic programming language, Parallel Alchemist cannot handle all the cases where shared memory is accessed. For instance, if a pointer variable is directed to a shared memory and the shared memory is modified through this pointer, the environment does not provide mutual exclusion.

The environment generates the declaration of the shared variables and their associated mutexes. These are declared as global symbols, accessible to any program module. The declaration is made in a separate initialization file. The shared memory and the mutexes are also declared as external symbols in an initialization header file, which is included in every module whose code is generated by the environment.

The initialization file also defines the main function. This function initializes the mutexes associated with the common shared variables and calls the **pmain** routine. The command line arguments, *argc* and *argv*, are passed to this routine as parameters. When pmain returns, the main function destroys the mutexes. The initialization and destruction of the mutexes are necessary because in certain parallel models, as the Mulplix native programming model, mutexes cannot be directly initialized in their declarations: the initialization must be made by a routine or system call.

Finally, the environment generates the batch command file to compile and link the code. This batch file has the format accepted by the make utility of the standard UNIX operating system. Information on objects and libraries that have to be linked together with the generated code helps on the batch command file generation.

7. Programming Examples

This section presents two programming examples under Parallel Alchemist. These examples show how the environment generates code and highlights some features described in the previous sections.

7.1. Inner Product

Table 2 presents a code segment generated by Parallel Alchemist, which implements the inner product between two vectors, ***first*** and ***second***. The result of the inner product is kept on a shared variable, called ***result***. The code was generated for the Pthreads model.

```
Double          first[10], second[10];
double          result;
pthread_mutex_t  mx_result;
...

void inner_product (int ord)
{
  int      i;
  double  acum = 0.0;

  for (i = 0; i < 5; i++)
    acum += first[ord +i] * second[ord+i];

  if (pthread_mutex_lock (mx_result) < 0)
    printf ("Lock error: mx_result \n");

  result += acum;

  if (pthread_mutex_unlock (mx_result) < 0)
    printf ("Unlock error: mx_result \n");
}
...
```

```
void pmain (int argc, char *argv[])
{
  int      j, tids[2];

  init_vectors ();
  result = 0.0;

  for (j = 0; j < 2; j++)
  {
    if (thr_create (&tids[j], 0, inner_product, j) < 0)
      printf ("Thread creation error: inner_product\n");
    if (0)
      pthread_detach (tids[j]);
  }

  for (j = 0; j < 2; j++)
    if (thr_join (tids[j]) < 0)
      printf ("Wait error: %d\n", tids[j]);

  printf ("Inner Product: %d\n", result);
}
...
```

Table 2 - Inner product code generated by Parallel Alchemist

The thread routine *inner_product* receives a single integer parameter, indicating which vector part it must handle: the upper half or the lower half. The code inside the routine is generated based on its programming blocks, except for the mutex lock and unlock, which are generated automatically by Parallel Alchemist. These code segments include a statement block, which accesses the common shared variable *result*. The accesses to the write-once shared variables *first* and *second* are left unprotected.

In the beginning of the code, the write-once shared variables *first* and *second* are declared, based on their type prefix (double) and suffix (vector of 10 doubles). The common shared variable *result* is also declared, with its associated mutual exclusion semaphore, *mx_result*.

The *pmain* routine initializes the vector using the *init_vectors* routine, which is not shown in the above code segment, but is also generated by the environment. Next, *pmain* assigns zero to the shared variable *result*. This access to the common shared variable is not protected, once the code generator detects that it occurs when only the main thread is running.

After the initialization, *pmain* creates two threads. Each thread calculates the inner product for one half of the vectors. The partial result is added to *result*. This access to the shared variable is protected by its mutex.

Finally, the inner product result is presented to the user. The environment checks the multithread safe routines list and detects that *printf* is among these routines. As the routine call is MT-Safe, the code generator provides no protection to the access of the shared variable result.

7.2.Gaussian Elimination

Table 3 presents another code segment generated by Parallel Alchemist. This code implements a gaussian elimination algorithm, using a 5 x 5 matrix. The matrix is represented by a write-once shared variable, called ***matrix***.

```
float       matrix[5][5];
int         lpivot;
MUTEX       mx_lpivot;
int         nowork;
MUTEX       mx_nowork;
...

void pmain (void)
{
  float result;
  int     j, ok;

  nowork = 0;
  result = 1;
  lpivot = 0;

  do
  {
    if (matrix[lpivot][lpivot] == 0)
    {
      change_line (lpivot);
      result = result * -1;
    }

    if (thr_spawns (4 - lpivot, thread, 0, 0) < 0)
      printf ("Synchronous creation error: thread
\n");

    if (nowork == 4 - lpivot)
    {
      ok = is_triangular ();
      if (ok) break;
    }

    lpivot++;

  } while (!(lpivot < 4));

  for (j = 0; j < 5; i++)
    result = result * matrix[j][ j];

  printf ("Result: %.3f\n", result);
}

...

void thread (int arg)
{
  int     j, line;
  float factor, pivot, *v;

  if (mx_lock (mx_lpivot) < 0)
    printf ("Lock error: mx_lpivot\n");

  line = lpivot + arg + 1;
  pivot = matrix[lpivot][lpivot];
  factor = matrix[line][lpivot];
  v = matrix[lpivot];

  if (mx_free (mx_lpivot) < 0)
    printf ("Unlock error: mx_lpivot\n");

  if (factor != 0.0)
  {
    for (j = 0; j < 5; i++)
      matrix[line][j] -= factor * v [j] /
                                      pivot;
  }
  else
  {
    if (mx_lock (mx_nowork) < 0)
      printf ("Lock error:mx_nowork\n");

    nowork++;

    if (mx_free (mx_nowork) < 0)
      printf("Unlock error: mx_nowork\n");
  }
}
...
```

Table 3 - Gaussian elimination code generated by Parallel Alchemist

The algorithm works in the matrix by considering a pivot row at each step. Matrix rows succeeding the pivot row are simplified by the pivot element, the pivot row element which belongs to the matrix main diagonal. The pivot row is represented by a common shared variable, called ***lpivot***.

The algorithm creates synchronous threads to handle the simplification of each matrix row after the pivot row. As the matrix is simplified, the algorithm uses another common shared variable, called ***nowork***, to verify if the matrix is already triangular.

As all threads are synchronously spawned, Alchemist's code generator detects that no protection is needed over the manipulation of shared memory on the *pmain* routine. The code generator provides mutual exclusion protection to accesses to *lpivot* and *nowork* within the *thread* routine.

The above code was generated for the Mulplix native parallel programming model. Mutexes are declared for the common shared variables *lpivot* and *nowork*. The shared variables are also declared.

Note that the value of *lpivot* is not modified within the *thread* routine. As all parallel accesses to this shared variable are read-only, it requires no mutual exclusion protection. Parallel Alchemist's actual code generator cannot detect read-only parallel accesses to shared memory, so it protects the manipulation of *lpivot* within the *thread* routine. However, this overhead can be avoided. If the user knows that this variable will only be used in read-only parallel accesses, the variable can be declared as write-once.

8. The Implementation

Parallel Alchemist is implemented in **Java** [FLA96], running on version 1.1.1, distributed by Sun Microsystems. It has originally been written for the Solaris platform, but inherits the portability of the Java language.

Figure 3 displays a normal operation screen within Parallel Alchemist. The main window is divided into two panels. The left panel presents the program structure, showing the shared variables, the modules and theirs associated routines. Once an object is selected on the left panel, the right panel presents its properties.

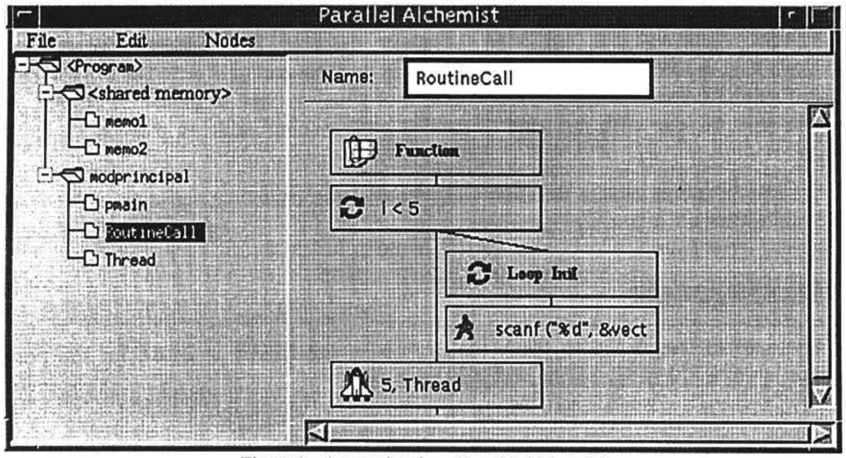

Figure 3 - A snapshot from Parallel Alchemist

When a shared memory is selected, the right panel presents its name, type prefix and suffix type. When a module is selected, the right panel presents its name and declarations.

When a routine is selected, the right panel presents its name and its behavioral graph. The graph is modified with the Nodes menu options, that allow the creation of new nodes, deletion of nodes and the edition of the node properties.

The properties of a node are edited in separated windows and presented on the node, for clarity. Node connections are automatically created by the environment. Special nodes, like loop start, true condition detection and false condition detection are used to enhance the behavioral graph building process.

Parallel Alchemist is designed for model specificators and common developers. The main activity of model specificators is the creation of meta-schemes for existing parallel models. These meta-schemes are inserted into the environment, enhancing it with different models to generate code. Model specificators must be experts on the programming models to be modeled.

Common developers are the final users of the environment. They build their programs graphically, using the Alchemist's development process, and generate their code in any parallel model available within the environment.

9. Conclusion

This paper presented Parallel Alchemist, a visual software development environment for shared memory based parallel programming models. The environment uses a parallel programming meta-model and meta-schemes of existing models to generate code in any available model.

Parallel Alchemist helps software developers with little expertise on parallel programming to create complex applications from simple programming blocks. Also, Parallel Alchemist warranties code portability across a wide range of parallel programming models. New shared memory based machine-specific parallel programming models can be used by the environment, since an experienced user can write a meta-scheme for those models.

Parallel Alchemist has not been originally designed for message passing parallel programming models, where the parallel activities on them are quite different from those of a shared memory based model. The development of a new version of Parallel Alchemist which will be able to cope with this kind of parallel programming model is planned for the near future.

We also intend to explore the ideas under Parallel Alchemist in association with the object oriented software development paradigm.

Acknowledgements

The authors would like to thank FINEP, CNPq and RHAE in Brazil for the financial support given in the development of this research work.

Bibliography

[AUD96] Aude, Júlio S. et al. "The Multiplus/Mulplix Parallel Processing Environment", Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks, Beijing, China, 1996

[AHM94] Ahmed, S.; Carriero, N.; Gelernter, D. "A Program Building Tool for Parallel Applications", DIMACS Workshop on Specifications of Parallel Algorithms, Princeton University, May 1994

[AZE93] Azevedo, Rafael P. "Mulplix: An UNIX-like Operating System for Parallel Programming", M.Sc. Thesis, COPPE/UFRJ, 1993, in portuguese

[BAR96] Barros, Márcio O. "Implementation of the Pthreads Model under the Mulplix Operating System", Technical Report NCE-01/96, NCE/UFRJ, 1996, in portuguese

[BOE76] Boehm, Barry W. "Software Engineering", IEEE Transactions on Computers, December 1976

[BRO95] J. C. Browne et al "Visual Programming and Debugging for Parallel Computing", IEEE Parallel and Distributed Technology, Vol. 3, No. 1, 1995

[CAR95] Carrera, Enrique V. et al. "P-RIO: Construção Gráfica e Modular de Programas Paralelos e Distribuídos", Proceedings of the VII Brazilian Symposium on Computer Architecture - Parallel Processing, Canela, Brazil, 1995

[FLA96] Flamagan, D. "Java in a Nutshell: A Desktop Quick Reference for Java Programmers", O'Reilly & Associates Inc., 1996

[GEI94] Geist Al, Beguelin A., Dongarra J., Jiang W., Mancheck R., Sunderam V., "PVM - A users guide and tutorial for Network Parallel Computing", The MIT Press, Massachusetts, 1994

[GRA95] Grahan, John R. "Solaris 2.x: Internals and Architecture", McGraw-Hill, Inc., 1995

[HIL92] Hils, D. D. "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", Journal of Visual Languages and Computing, Vol. 3, No. 1, 1992

[IEE94] Institute for Electrical and Electronic Engineers, POSIX P1003.4a, "Threads Extension for Portable Operating Systems", 1994

[ING88] Ingalls, D. et al. "Fabrik, A Visual Programming Environment", OOSPLA'88 Proceedings, ACM Sigplan Notices, Vol. 23, No. 11, November 1988

[KAR95] Karsai, Gabor. "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming", IEEE Computer, March 1995

[KER88] Kernighan, B. and Ritchie, D. "The C Programming Language", Second Edition, Prantice-Hall Software Series, 1988

[MPI95] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", MPI Forum Draft, June 1995

[NEW95] Newton, P. and Dongarra, J. "Overview of VPE: A Visual Environment for Message-Passing", Heterogeneous Computing Workshop at the 9'th International Parallel Processing Symposium, Santa Barbara, EUA, 1995

[PRE92] Pressman, Roger. "Software Engineering: a Practitioner's Approach", Third Edition, McGraw-Hill International Editions, 1992

[REP95] Repenning, A.; Sumner, T. "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages", IEEE Computer, March 1995

[SCA93] Schaeffer, J et al. "The Enterprise Model for Developing Distributed Applications", IEEE Parallel & Distributed Technology, Vol. 1, No. 3, August 1993

[SUN95] Sun Microsystems, Inc. "Multithreaded Programming Guide", 1995