Multithread Object-Oriented Langauge EusLisp, for Parallel andAsynchronous Programming in Robotics

Toshihiro Matsui (matsui@etl.go.jp)
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibaraki 305, Japan

Abstract: *EusLisp is an object-oriented programming language with geometric modeling facilities. In robot programming pursued in EusLisp, parallel programming capability is claimed to gain more computation power and to facilitate asynchronous programming of various devices. Recent workstations are beginning to have multi-processors, on which users can easily build parallel programs making use of operating systems multi-threading capability. The shared-memory parallel programming model provided by the threads is suitable for parallel Lisp which manipulates pointers to shared objects. Implementation of multithreading Lisp is, however, not easy since memory-sharing complicates the automatic memory management, and performance could be degraded by mutual exclusion and synchronization. This paper presents EusLisp's parallel programming facilities on Solaris, the implementation of the memory manager, and a performance analysis of parallel programs running on a real multi-processor machine.*

1. Introduction

EusLisp[1,2] is an object-oriented Lisp whose goal is an implementation of a 3D geometric modeler and its application to high-level robot programming. It has been applied to many areas of robotics research such as collision free path planning, grasp planning, assembly planning, analysis of motion in contact, simulators for teleoperation, etc. In order to extend the application fields in a more real-time oriented direction, we redesigned EusLisp to support parallel and asynchronous programming using the Solaris 2 operating system's multithread facility.

Asynchronous programming is required for programs to respond to external events via multiple sensors occurring independently of the program's state. Parallel programming is effective to improve performance of computation bound processing such as image processing and interference checking in path planning.

2. Implementation of Multithread EusLisp

2.1 Multithread in Solaris 2

Our new Multithread EusLisp (MT-Eus) runs on the Solaris 2 operating system with one or more processors. Solaris's threads are units for allocating CPU in a traditional UNIX process, having shared memory and different contexts[4]. The thread library provided by the Solaris OS allocates each thread to a single LWP (light weight process), which is a kernel resource. The Unix kernel schedules the allocation of LWPs to one or more physical CPUs based on thread priorities assigned to each thread. **Fig.1** depicts the relations between threads, LWPs, and CPUs. We made two major changes in the design of the contexts and the memory management of EusLisp to upgrade it to multithread capabilities.

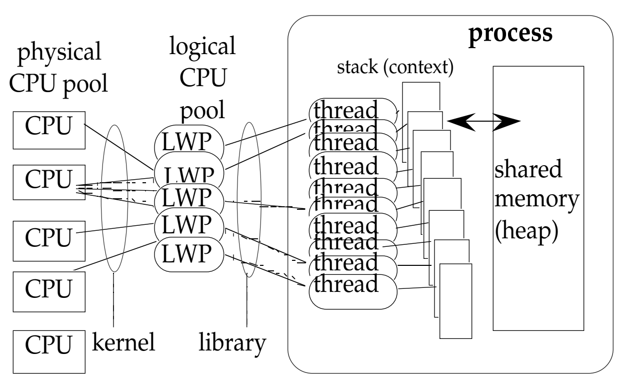

Fig. 1 Threads in a process and allocation of CPU

2.2 Context Separation

MT-Eus allocates private stacks and contexts to each threads so that they can run independently of each other. Objects such as symbols and conses are allocated in the shared heap memory as in sequential EusLisp. Therefore, thread-private data such as block labels, catch tags, and local variables are protected from other threads, whereas values (objects) pointed by global variables are visible to all threads allowing information exchange among threads.

A context consists of a C-stack, a binding-stack, and frame pointers that chain lexical blocks such as lambda, block, catch, let, flet, and so on, and is established when a new thread is created. Since more than one context can be active at the same time on a real multi-processor machine, we cannot hold a single pointer to the current context in a global variable. Rather we have to add one more argument to every internal function to transfer the pointer from the topmost eval to the memory manager at the bottom.

2.3 Memory Management

EusLisp adopts a *Fibonacci buddy* memory management scheme[1,3] in a single heap for every type of object. After running programs having different memory request characteristics, we have been convinced that Fibonacci buddy can allocate objects of various sizes equally fast, garbage-collects quickly without copying , and exhibits high memory utilization (internal loss is 10 to 15%, external loss is negligible). For multithreading, the second point, i.e., non-copying GC, is very important. If addresses of objects were changed by copying-GC, pointers in the stack and CPU registers of all thread contexts would have to be redirected to new locations, which is impossible or very difficult.

All memory allocation requests are handled by the `alloc` function at the lowest level. `Alloc` does mutex-locking because it manipulates the global database of free lists. Since we cannot predict when a garbage collection begins and which thread causes it, every thread must prepare for sporadic GCs. All pointers to living objects have to be arranged to be accessible by the GC anytime to prevent them from being reclaimed as garbage. This is done by storing the pointers to the most recently allocated objects in fixed slots of each context, instead of trusting they are maintained on the stacks.

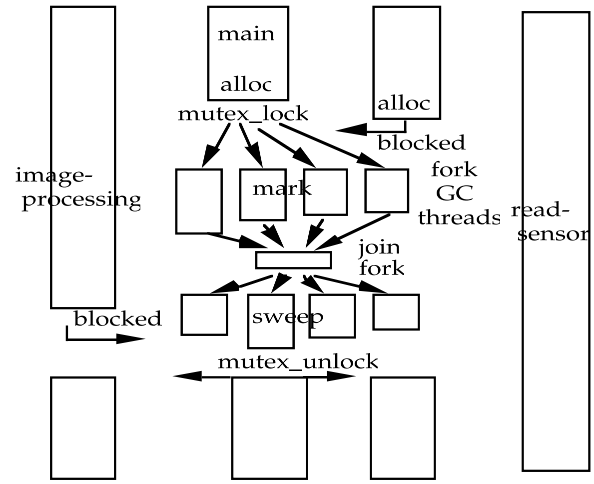

Fig. 2 threads when the GC is invoked: some threads can run in parallel with the GC, while the GC runs several threads in parallel.

Fig. 2 illustrates flow of threads requesting memory and forked inside GC to process marking and sweeping in parallel. Note that threads that do not request memory or manipulate pointers can run in parallel with the GC, improving real-time response of the low-level tasks such as signal processing and image acquisition.

3. Asynchronous and Paralle Programming Constructs

3.1 Thread Creation and Thread Pool

In order for Solaris to execute a program in parallel on many processors, the program needs to be written as a collection of functions, each of which is executed by a thread dynamically created in a process. Although the time required for thread creation is faster than process creation, it takes a few milli-seconds for EusLisp to start off a thread after allocating stacks and setting a page attribute for detecting stack-overflow. Since this delay, which should be compared to a function invocation, is intolerable, sufficient number of threads are created by the `make-thread` function beforehand and put in a thread pool, eliminating the need for system calls at evaluation time. Each thread in the thread pool is represented by a thread object consisted of thread-id, several semahores for synchronization, and slots for argument and evaluation result transfer.

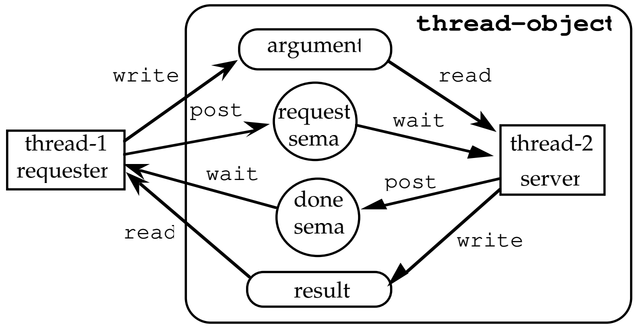

Fig. 3 Thread object and synchronization for parameter/result exchange

3.2 Parallel Execution of Threads

For the allocation of parallel computation to threads, the `thread` function is used. `Thread` takes one free thread out of the thread pool, transfers arguments via shared memory, wakes up the thread by signalling the semaphore as indicated in **fig. 3**, and returns a thread object to the caller without blocking.

The woken-up thread begins evaluation of the argument running in parallel to the calling thread. The caller uses `wait-thread` to receive the evaluation result from the forked thread. The `plist` macro is a more convenient form to describe parallel evaluation of arguments. `Plist` attaches threads to evaluate each argument and lists up results after waiting for all threads to finish evaluation.

3.3 Synchronization primitives

MT-Eus has three kinds of synchronization primitives, namely *mutex locks*, *condition variables*, and *semaphores*. *Mutex locks* are used to serialize accesses to shared variables between threads. *Condition variables* allow a thread to wait for a condition to become true in a mutex-locked section by temporarily releasing and reacquiring the lock. *Semaphores* are used to inform occurrences of events, or to control sharing of a limited number of resources. These synchronization primitives cause voluntary context switching, while the Solaris kernel generates involuntary task switching on a time-sliced scheduling basis.

3.4 Barrier synchronization

Barrier-synch is a mechanism for more than two threads to synchronize at the same time. For this purpose, an instance of the `barrier` class is created and threads that participate in the synchronization register themselves in the object. Then, each thread sends the `:wait` message to the barrier object, and the thread is blocked. When the last thread registered in the object sends its `:wait` message, the waits are released and all waiting threads get a return value of `T`. *Barrier-sync* plays an important role of global clocking in a multi-robot simulation.

3.5 Synchronized memory port

Synchronized memory port is a kind of stream to exchange data between threads. Since all threads in a process share the heap memory, if one thread binds an object to a global variable, it instantly becomes visible to other threads. However, shared memory lacks capability to send events that the global data is updated. *Synchronized memory port* ensures this synchronization for accessing a shared object. A synchoronozed memory port object consists of one buffer slot and two semaphores used for synchronizing read and write.

3.6 Timers

Real-time programs often require functions to execute at predetermined timing or to repeat in particular intervals. Sequential EusLisp could run user's functions triggered by signals generated periodically by unix's interval timers. This preemption can cause deadlock in MT-Eus, because interruption may occur within a mutexed block. Therefore, control must be transferred at secured points such as at the beginning of eval. To avoid delays caused by the above synchronization, MT-Eus also provides signal-notification via semaphores. In other words, the `signal` function takes either a function or a semaphore that is called or posted upon the signal arrival. Since the semaphore is posted at the lowest level, latency for synchronization is minimal.

Fig. 4 shows a virtual image processing program coded by using the features described so far. Image input thread and filtering threads are created. samp-image takes image data periodically by waiting for samp-sem to be posted every 33msec. Two threads synchronize via read-and-write of a thread-port. Filter-image employs two more threads for parallel computation of filtering.

```
(make-threads 8)
(defun samp-image (p)
   (let ((samp-sem (make-semaphore)))
     (periodic-sema-post 0.03 samp-sem)
     (loop (sema-wait samp-sem)
           (send p :write (read-image))))
(defun filter-image (p)
  (let (img)
     (loop (setf img (send p :read))
           (plist (filter-up-half img)
                  (filter-low-half img)))))
(setf port (make-thread-port))
(setf sampler (thread #'samp-image port))
(setf filter (thread #'filter-image port))
```

Fig.4 Example of Thread Programming

4. Measured Parallel Gains

Table. 1 shows the parallel execution performance measured on a Cray Supserserver configured with 32 CPUs. Linear parallel gain was obtained for the compiled fibonacci function, because there is no shared memory access and the program code is small enough to be fully loaded onto the cache memory of each processor. Contrally, when the same program was interpreted, linearly high performance could not be attained, since memory access scatters. Further, some programs that frequently refer to shared memory and request memory allocation cannot exhibit better performance than a single processor execution. This can be understood as the result of frequent cache memory purging.

		parallel gain				GC (ratio)
	processors	1	2	4	8	
(a)	fibonacci (compiled)	1.0	2.0	4.0	7.8	0
(b)	fibonacci (interpreted)	1.0	1.7	2.7	4.4	0
(c)	copy-seq	1.0	1.3	0.76	0.71	0.15
(d)	make-cube	1.0	0.91	0.40	0.39	0.15
(e)	interference-check	1.0	0.88	0.55	0.34	0.21

Table 1 Performance of parallel execution

5. Future Project

In our autonomous mobile robot project, we plan to develop a multi-robot simulator, human-robot interaction navigator, visuo-audio event locator, human motion recognizer, etc, all of which are supposed to find synchronization between events occuring seemingly asynchronously, making use of prior knowledge about the objects. We believe MT-Eus's asynchronous programming and geometric modeling facilities will play essential roles there.

Although multithreading is a powerful and inevitable tool for robot programming, it also brings annoyances: programmers always have to worry about mutex of global variables, synchronization, exceptions, and so on. Among them, the exception handling seems to be the toughest for interactive programming languages like EusLisp. When an error is reported from one of the threads running in parallel, it is hard to tell which thread caused the error, which thread your ^C is interrupting, which thread is requesting key input, etc. At least a sophisticated parallel programming environment and parallel program analyzer using multiple windows is required.

Reference

1) Matsui, T. and Inaba, M : EusLisp an Object-Based Implementation of Lisp", Journal of Information Processing, 13, 3, (1990).
2) Matsui, T.: EusLisp verson 7.27 Reference Manual, Electrotechnical Laboratory Technical Report, ETL-TR-92-36, (1992).
3) Cranson, B. and Thomas, R.: A Simplified Recombination Scheme for the Fibonacci Buddy System, Communication of ACM, 18, 6, (1975).
4) ---, Multithreading, SunOS 5.3 System Services, pp. 105-134, Sun Soft, (1994).
5) Halstead, R. H. Jr. : Implementation of Multilisp: Lisp on a Multiprocessor, Conference record of the 1984 ACM symposium on a Lisp and functional programming, pp. 9-17 (1984).