Top Level, Inc.
When you're ready for the future™

Parallel Common Lisp for the 90's

The increase in computational resources required for today's more ambitious and real-time oriented applications is making effective utilization of multiprocessor computer systems essential. The current generation of shared-memory multiprocessors offers the potential for an order-of-magnitude increase in computational power over single-processor systems. A critical component in achieving this potential is a parallel programming language that is both efficient and easy to use.

Top Level Common Lisp (TopCL) is an extended Common Lisp designed and implemented specifically for the effective utilization of parallel computer systems. Based on research at the University of Massachusetts at Amherst, TopCL is now commercially available and supported through Top Level, Inc. Top Level Common Lisp is currently available on the Encore Multimax and Sequent Symmetry shared-memory multiprocessors.

Three key capabilities of TopCL make it especially suited for exploitation of parallel computers. As with a Common Lisp, TopCL provides an extensible programming language that allows higher-level parallel constructs to be defined that are appropriate for particular types of problems. The run-time type-checking of Common Lisp readily allows the use of the *future* construct, providing implicit synchronization of parallel activity. Finally, TopCL provides four grain sizes of parallel operators. This promotes efficiency by allowing both fine-grained parallelism when certain capabilities are not needed, as well as larger-grained parallelism when more capabilities (with their associated overheads) are required.

The Future of Common Lisp

The use of parallelism to solve a problem can be characterized by three components: (1) breaking up a task into smaller pieces that can be performed in parallel (the fork), (2) insuring these pieces have the necessary resources to do the work, and (3) combining their results (the join). Since TopCL provides for user-specified parallelism, the programmer is responsible for deciding when to fork. The type of fork used specifies what kind of resources are needed. A join occurs *implicitly* through the use of *future* objects.

When a computation is forked, it is represented by (referenced through) a future object. Until its computation completes, the future object is *undetermined*. Once the computation completes, the future object "becomes" the value returned. The future is now *determined* and is consequently indistinguishable from this value.

When a *strict* operation encounters a future object, it implicitly *forces* the future object to be determined, and will wait until the value becomes available before continuing. Strict operations involve looking at the actually value or data-type of an object. For example, addition requires its arguments to be numbers. Since this synchronization occurs implicitly (below the level of the language), existing programs can use future objects without any knowledge that computation is occurring in parallel.

Through use of the future construct, TopCL provides a natural and powerful way to deal with the added complexity of parallel programming.

From Small-Grained to Large-Grained

To use parallelism effectively, the associated overhead must not be prohibitive. If it takes more than two seconds to perform two one-second computations in parallel, there is clearly no advantage in doing so. To avoid this, the overhead in using parallel computation should be as small as possible. However, if a pair of computations takes a long time, proportionately more overhead can be incurred to run them in parallel and still gain performance improvements.

TopCL defines four different fork operators with different overheads and properties. These are the Thread, the Task, the Process, and the Node.

196 North Pleasant Street • Amherst, Massachusetts 01002 • (413) 256–6405

The Thread is the lowest-overhead fork provided. In the current implementation, a thread has a creation overhead of approximately 8 function calls[1]. A thread is assumed to be a short computation, and therefore the system may choose not to compute it in parallel if no free computing resource is available.

The next level of fork is the Task. Its creation overhead is roughly 300 function calls. It has its own control stack and can therefore be time-shared as well as executed in parallel. It also maintains its own binding stack and therefore enjoys a more well-defined computational context.

The next level of fork is the Process. Its creation overhead jumps to roughly 40,000 function calls. A process is a task with additional capabilities, such as maintaining its own free-storage pool, handling interrupts, and calling out to external routines.

The largest level of fork is the Node. Its creation overhead jumps to 750,000 function calls. A node maintains its own address space, and thus any objects accessible to a node must be *copied*. This allows node-level parallelism to potentially utilize an entire computing network. A node can function as either a single computation or as a computational server.

Obtaining Real Performance Improvement

To demonstrate the potential for performance improvement, a number of benchmark programs were modified to run in TopCL. The Fibonacci function was modified to compute its arguments in parallel using threads. The run-time improved sixfold using 10 processors. The TAK function using threads showed a 2-3 times speedup with 4 processors. The Triangle benchmark demonstrated a 12 times speedup using 16 processors.

While such small benchmark programs provide a common base for understanding and demonstrating potential, the important question is whether parallelism is applicable in real-world programs. One such program immediately available was the TopCL compiler itself. The compiler was parallelized using a three-process pipeline. One process reads forms from the input file, a second compiles them, and a third assembles and writes the output file. Using this strategy, the compiler runs up to three times faster than is possible with a single-processor system. Typical results show it cutting compile time in half. We have also used TopCL to make parallelism available to users of the Generic Blackboard System (GBB) In an example vision application, the parallel GBB program achieved fairly linear speedup, leveling out at just over an eightfold speedup with 12 processors. Experiments with a parallelized OPS5 program have showed 5 to 10 time speedups compared with single processor times.

The Best of the Lisp World

The development and debugging capabilities possible with a Common Lisp system are widely recognized as unmatched by any other programming environment. These capabilities are critical for the success of a parallel programming system.

The TopCL system was developed using a Texas Instruments Explorer Lisp Machine. Leveraging on this environment, Top Level, Inc. provides a powerful debugging and development interface to TopCL. Also ported to the Symbolics Lisp Machines, the interface allows a developer to pause and continue all computation on the multiprocessor. Inspector windows can be used to look at backtraces of all existing tasks, determine what future objects are being waited for, as well as inspect any lisp object. All through a point–click–drag interface. With our support of X-windows through CLX, this same level of interface will soon be available using a low-cost X-station.

Top Level also supports the Common Lisp Object System (CLOS), which defines a very powerful object-oriented programming model that can substantially reduce software development time.

The superior development environment possible with a Common Lisp system, combined with the efficient, extensible, and powerful parallel constructs of TopCL can bring the power of a multiprocessor to your most demanding applications.

[1] A function-call unit is the time needed to call a function with no arguments, and return with no useful value.

Top Level, Inc.

When you're ready for the future™

Top Level Common Lisp™
A Parallel Common Lisp

While others have been talking about the potential of parallelism, we've been working to make it a reality.

Three years of dedicated research and development at the University of Massachusetts at Amherst have culminated in the first high-performance parallel implementation of Common Lisp for shared-memory multiprocessors. With the commercial availability of Top Level Common Lisp, we've raised the Common Lisp standard to a new level.

Top Level Common Lisp was designed and implemented specifically to support efficient use of parallel computer systems. Integrating both distributed *and* shared-memory parallelism, Top Level Common Lisp provides four different levels or grain-sizes of parallel operators, allowing full utilization of the potential parallelism of an application. Each level uses the *future* construct, which provides *implicit* synchronization of parallel activity. Unlike systems that "graft" futures on top, Top Level maintains high performance for serial programs by supporting futures at the lowest level of the implementation.

But Top Level Common Lisp is more than just a parallel programming language. It also has advanced features you have come to expect from a Lisp system, such as a full editing command interface, a powerful debugger, an inspector, an interface to C programs, a fully developed module facility, and many more features. Top Level, Inc. is committed to the X-window standard, and is will provide a robust windowing environment that will match the high level of functionality available today with current Lisp Machines – one that can be used for both development *and* delivery.

And if you have already invested in the Texas Instruments Explorer or Symbolics Lisp Machine, Top Level Common Lisp has even more benefits. Instead of compromising your development environment, Top Level, Inc. *advances* the state of the art by using existing Lisp Machines to provide a powerful debugging and development interface to Top Level Common Lisp.

The combination of this environment with the computational resources of Top Level Common Lisp on the Encore or Sequent multiprocessor delivers an unprecedented high performance Lisp system.

196 North Pleasant Street • Amherst, Massachusetts 01002 • (413) 256–6405

Top Level, Inc.
When you're ready for the future™

Top Level Common Lisp
Parallel Common Lisp for the Real World

Modern Lisp Systems

From its beginnings in the 1950s, Lisp has evolved from a research tool found in university laboratories to a powerful and efficient language with a standardized dialect in Common Lisp. Its superior development environment can dramatically reduce development time compared with traditional programming languages. With the recent standardization of the Common Lisp Object System, a Common Lisp system provides one of the most powerful and productive programming languages available today. Advances in compiler technology and an emphasis on speed now yield Common Lisp implementations that are fast enough to be used as delivery vehicles for many software projects. With the ever-increasing speeds and memory capacities of modern computer systems, the small performance advantage of traditional programming languages pales in comparison with the superior development environment of a Common Lisp system.

Parallel Programming

Despite the seemingly endless advances in processor technology, the speed of light imposes a real limitation on processor speeds. As this limitation is approached, it is inevitable that some form of parallel computation must be employed to produce faster programs. While characteristics of parallel architectures can vary widely, the two basic architectures are shared-memory multiprocessors and distributed-memory multicomputers. A shared-memory architecture eliminates explicit sharing and provides a model of computation closer to existing serial systems, and thus appears most promising for immediate exploitation. Although a shared-memory programming model is easily exploitable, a distributed-memory system can provide vast computational resources. From networks of workstations to hundreds-of-processor multicomputers, the potential of these distributed resources cannot be ignored. Therefore Top Level Common Lisp is designed for *both* shared- and distributed-memory parallel programming. With the integration of shared- and distributed-memory parallel computing, Top Level Common Lisp promises to finally give the programmer the full computational power available in a computing environment.

However, early experience has shown that writing and debugging parallel programs can be more difficult than with serial versions. A good programming environment and language for supporting parallel programming is essential for effective utilization of multiprocessor systems. The superior development environment and extensibility of Common Lisp makes it the clear choice for use in a multiprocessing system.

The Future of Common Lisp

There have been many proposals for the parallelization of Lisp. One of the most powerful ideas is that of *futures*. When a fork is performed, the fork operation immediately returns a future object that will eventually receive the result of the computation. This future object can be used freely as any other object in the system while its value is being computed. If at any time the value of the future object is required, the operation waits for the computation to complete. The run-time type-checking inherent in Lisp systems makes this possible since objects can be manipulated without regard to their type. The future mechanism

196 North Pleasant Street • Amherst, Massachusetts 01002 • (413) 256–6405

is a powerful construct that is completely transparent to a program, providing *implicit* and automatic synchronization not possible with traditional, strongly typed languages.

In addition to the future mechanism, Top Level Common Lisp supports four different levels or grain-sizes of parallel constructs, each with different capabilites and overhead. This allows full utilization of the parallel potential of a program.

Finally, the ability to extend Lisp provides an important capability for a parallel language. Common parallel constructs can be defined with complex macro definitions. These macros expand into appropriate lower-level code, freeing the programmer from lower-level details. In fact, macros allow programmers to essentially construct their own parallel programming language using the parallel primitives provided by Top Level Common Lisp.

A Parallel Common Lisp

Until Top Level Common Lisp, the power of Common Lisp systems had been restricted to serial implementations. Anyone needing or wanting to exploit parallel programming has been forced to struggle with other languages, use slow simulation or interpreter systems, or has been told to wait ... and wait. With the commercial availability of Top Level Common Lisp, the future is finally here.

Top Level Common Lisp is a complete and high-performance implementation of Common Lisp. Unlike previous Lisp implementations, it was designed from the start for efficient execution of parallel Lisp programs. Futures are supported at the lowest level of the implementation, unlike the strategy of "grafting" them on top, which slows down programs that don't use futures. In addition, our compiler uses parallelism to run up to *three times* faster than is possible using a serial Lisp.

Unprecedented for Lisp systems, our debugger runs *outside* the address space being debugged — reducing behavior changes introduced by debugging and eliminating *debugger deadlock* which occurs when an internal debugger requires resources or locks held by the tasks being debugged. By allowing our debugger to run outside the multiprocessor, Top Level provides a development interface that runs on either a TI Explorer or Symbolics Lisp Machine, protecting your existing investments in programmer productivity.

Top Level Common Lisp is more than just a parallel Common Lisp system. It also has the features you expect in modern Lisp systems, such as an optimizing compiler, powerful debugger, tracing and stepping facilities, an interface to C code, a fully developed module construct for manipulating program files, an inspector that allows the display and updating of data structures, object-oriented capabilities, and X-windows support.

To find out more about how Top Level can make parallelism work for you, give us a call at (413) 256-6405.