Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems

Milind Girkar, Microcomputer Research Labs, Santa Clara, Intel Corporation
Mohammad R. Haghighat, Microcomputer Research Labs, Santa Clara, Intel Corporation
Paul Grey, Microcomputer Research Labs, Santa Clara, Intel Corporation
Hideki Saito, CSRD, University of Illinois
Nicholas J. Stavrakos, CSRD, University of Illinois
Constantine D. Polychronopoulos, CSRD, University of Illinois

Index words: parallelism, Windows NT*, fibers, compilers, multithreading.

Abstract

Powerful desktop multiprocessor systems based on the Intel Architecture (iA) offer a formidable alternative to traditional scientific/engineering workstations for commercial application developers at an attractive cost-performance ratio. However, the lack of adequate compiler and runtime library support for multithreading and parallel processing on Windows NT* makes it difficult or impossible to fully exploit the performance advantage of these multiprocessor systems. In this paper we describe the design, development, and initial performance results of the Illinois-Intel Multithreading Library (IML), which aims at providing an efficient and powerful (in terms of types of parallelism it supports) API for multithreaded application developers. IML implements a parallel execution environment, which creates, enqueues, dequeues, binds, and schedules user-level threads on Windows NT* threads and fibers. One of the unique and novel features of IML is its support for both loop-level (data) parallelism and task-level (functional) parallelism, as well as nested parallel threads. Although loop-level parallelism is most useful in scientific and engineering applications, functional parallelism is often the norm in multimedia, Internet, and Java* applications. IML upgrades the multithreading support available on the iA-based Windows NT* platforms to levels comparable or superior to those found on high-end parallel systems and supercomputers. Multithreading a number of diverse benchmarks (ranging from POV-Ray to SPECfp95 and the BLAS routines) using IML resulted in significant speedups on a 4-way SMP Pentium® Pro processor based system. Future releases of IML will support several loop scheduling schemes as well as controlled thread migration for the purpose of dynamic load balancing. The programmer or the compiler would thus be able to customize scheduling on a per loop basis taking into consideration performance-sensitive characteristics such as branches inside loops and data locality. The Intel C/C++ and FORTRAN compilers and the Parafrase-2 experimental parallelizing compiler are being enhanced in order to automatically generate the IML API, thereby facilitating the development of multithreaded application codes that fully exploit the performance potential of iA-based multiprocessor servers and desktops.

Introduction

Parallel processing is rapidly becoming mainstream technology influencing architecture and software design from the home PC market (in the form of instruction-level parallelism (ILP), Intel MMX™ technology, and multiple processors on PC boards) to the business field where Symmetric Multi-Processing (SMP) servers have become increasingly popular. While Intel compilers provide intrinsics to generate Intel MMX instructions so that independent software vendors (ISVs) can easily incorporate this technology into their products, there has been little support for programmers to make use of iA-based SMP systems for parallel processing. In fact, multiprocessing may be the most significant enabling factor for moving large-scale engineering and business applications to iA-platforms for the first time, thereby opening new opportunities in the discriminating high-end market.

The Illinois-Intel Multithreading Library (IML) upgrades the multiprocessing support on iA-based Windows NT* to levels comparable to or higher than the multiprocessing libraries provided for high-end multiprocessor servers and scalable parallel processor systems.

IML, unlike other previous or current runtime systems, supports functional parallelism [2][3][4] where task execution conditions are expressed by a directed acyclic graph (DAG), in addition to the more conventional *loop* parallelism and the single-level cobegin/coend functional parallelism. IML is also capable of exploiting *arbitrarily nested parallelism*, which has not been available in any of the commercial multiprocessing libraries, including high-end multiprocessors from Sun Microsystems and SGI, as well as supercomputer systems from Cray, Fujitsu, NEC, and IBM. SPMD-style nested parallelism is an optional feature of the OpenMP standard [6].

IML has been used at the Center for Supercomputing Research and Development (CSRD) and Intel for in-house software development. Application programs written in FORTRAN, C, and C++ for numerical computing, database, and 3D graphics have been successfully ported to the IML library. The Parafrase-2 automatic parallelizing compiler [7] developed at the University of Illinois has been modified in order to generate calls to IML automatically, thus exploiting loop and functional parallelism exposed by the compiler. The IML binaries and the documentation are now available to the public through the IML home page on the web [5]. The Intel compilers are being modified to automatically parallelize programs and generate calls to the IML library.

The rest of the paper describes the design and the implementation of IML and the results from our initial performance study. Our measurements indicate that the performance of IML matches or exceeds highly-tuned commercial libraries for existing multiprocessors for many common single-level DOALL loops while adding support for more general parallelism. Conventional libraries provide multithreading support for simple, singly-nested parallel loops, which allows these libraries to be simpler in design and to incur lower overhead costs. IML implements a queue-based multithreading environment, which supports general loop and functional parallelism and allows arbitrary nesting of parallel loops and unstructured parallel constructs such as nested cobegin/coend.

Design

Basic Design Alternatives: Single Parallel-Task Descriptor vs. Pool of Parallel Tasks

Existing commercial and experimental multiprocessing libraries allow only one parallel loop to be executed at a time. If a second parallel loop is encountered during the execution of a parallel loop (as is the case with nested parallel loops), the second loop is treated as sequential. The same can be said for nested cobegin/coend constructs, also referred to as functional parallelism. Such an execution environment can be supported by a single task descriptor specifying the loop body and the number of iterations. However, supporting the execution of multiple parallel loops (which may be nested or disjointed in arbitrary control flow patterns) or functional parallelism (where precedence requirements are specified by a DAG) necessitates a pool of ready parallel tasks from which assignments to user threads are made (Figure 1). In IML, a collection of task queues[1] is used to implement such a pool of parallel tasks. User-level threads such as one or more iterations of a parallel loop or a function call are then bound to ready-to-execute tasks and are scheduled for execution. In Figure 1, threads execute the task scheduling loop *fetch-execute-enqueue* until the program terminates.

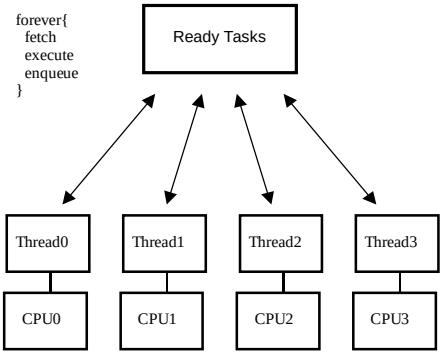


Figure 1: Pool of parallel tasks

Queue-Based System Design Alternatives: Centralized Queue, Distributed Queue, or Global-Local Queue

A pool of parallel tasks can be implemented by a single shared queue. Two major drawbacks of this approach are contention and locality. In order to exploit maximum parallelism, thread parallelism should be exploited at the finest possible granularity that amortizes the overhead of task management and scheduling. However, small task sizes lead to more scheduling events, consequently increasing the contention on a shared queue. Moreover, in cache-coherent systems, this is likely to lead to poor cache hit rates as a result of multiple processors updating a single queue structure. Alternatively, multiple queues can

[1] Throughout the paper, the term queue is used in a broad sense, a list that is subject to insertions and deletions.

be distributed among threads, for example, one queue for each thread. In this configuration, if a thread cannot find more work on its local queue, it can access remote queues, which facilitates load balancing. When workload is well distributed among tasks, contention is minimized which, in turn, promotes cache locality of the task queues. An extreme drawback to this approach can arise when an idle thread accesses a large number of remote queues before finding a task to execute[2]. The shortcoming of the two approaches discussed above can be eliminated by introducing a global queue, in addition to local queues. IML employs a distributed queue configuration because the primary target architecture consists of a small number of processors. Future releases may provide generalized support for large-scale parallel or distributed computing systems.

Implementation Basic API Functions

Application programmers can write parallel programs with the following functions. Support for the OpenMP standard [6] is currently being implemented.

- **`iml_DOALL()`** enqueues a parallel loop task. This function returns when the loop task is completed. The parameters of this function specify the pointer to the function representing the loop body, the number of iterations, the policy for loop scheduling,[3] the minimum chunk size, the number of parameters to the loop body, and the actual parameters to the loop body.
- **`iml_COBEGIN()`** enqueues a set of functionally parallel tasks. This function returns when all the tasks are completed. The parameters of this function specify the number of tasks, the pointers to the functions representing the tasks, the number of parameters to the tasks, and the actual parameters to the tasks.
- **`iml_EnQ()`** enqueues a task that is not a parallel loop. This function returns immediately after enqueueing the task, and thus does not wait for the completion of the task. The parameters of this function specify the pointer to the function representing a task, the number of parameters to the task, and the actual parameters to the task. This interface allows programmers to implement arbitrary functional parallelism.
- **`iml_DecAndFetch()`** performs user-level synchronization. This function atomically decrements the counter and returns the value after the decrement. Combined with **`iml_DOALL()`**, this can be used to implement a DOACROSS (partially parallel) loop. Combined with **`iml_EnQ()`**, this can be used during the scheduling of DAG-parallel tasks. This function takes the pointer to a counter as its argument.

Extended API Functions

Extended API functions are provided for experienced application programmers. The extended API can also be used by a compiler for automatic generation of calls to IML (such is the case with Parafrase-2 and Intel parallelizing compilers). Simple examples of the basic and the extended API functions are given in the Appendix.

- **`iml_ReInitMultiThread()`** changes the number of active threads used by IML.
- **`iml_GetThreadID()`** returns the thread ID of the current thread.

The full performance potential of the above API can only be exploited with appropriate support from the OS kernel. In particular, an application can add/release threads as it goes through different phases of its execution in a way that accurately reflects the parallelism in the underlying computation. This results in better utilization of processors and memory and translates not only to lower execution times, but also to improved average workload turnaround time in a multi-user environment. OS support at the level of allocating and reclaiming resources from user processes would be necessary in order to exploit this capability of IML.[4] However, this is not the case with Windows NT* at present.

Windows NT*: Threads and Fibers

A thread is a unit of computation scheduled by the operating system to run on a processor. A fiber is a unit of computation that runs on a thread and is scheduled by a user[10]. Some of the important characteristics of fibers are as follows:

- Fiber switching is measured to take 50-60 cycles on a 200 MHz Intel Pentium® Pro processor. On the other hand, the cost of suspending a thread is orders of magnitude greater.
- There is an order of magnitude difference in the cost of creation and deletion between fibers and threads.
- Similar to threads, fibers provide a user-level context that includes a program counter, registers, and a stack.

[2] Effective scheduling algorithms and thread migration schemes minimize the occurrence of such extremes.

[3] IML implements various scheduling algorithms from which the programmer can select on a loop-by-loop basis.

[4] Hybrid implementations are possible but cumbersome and may conflict with software compatibility.

- Scheduling of fibers is controlled by the user, while scheduling of threads is controlled by the operating system.
- The relationship of fibers to threads is analogous to that of threads to processors. An active fiber is bound to a thread, just as an active thread is bound to a processor. A thread can have at most one active fiber; similarly, a processor can have only one active thread. *Inactive* (or unbound) threads and fibers do not receive any computational resources.

Figure 2 illustrates the relationship between threads and fibers. Without fibers, a context switch, even within a process, is performed by the operating system. In Figure 2(a), bold lines represent the bindings between threads and processors. A context switch corresponds to a reconnection of a bold line from one thread to another thread. This operation has two drawbacks. It is expensive and not controllable by the user. These drawbacks can be overcome with the introduction of fibers. Fibers detach execution contexts from threads, allowing their scheduling to be explicitly controlled by the user. Multiple threads are still needed to maintain multiple active fibers. In Figure 2(b), regular lines represent the bindings between fibers and threads. A user-level context switch corresponds to a reconnection of a regular line from one fiber to another fiber, while a kernel-level context switch is still represented by a reconnection of a bold line. Since fibers are lightweight, easy to manage, and can be explicitly scheduled by the user, they are used in the

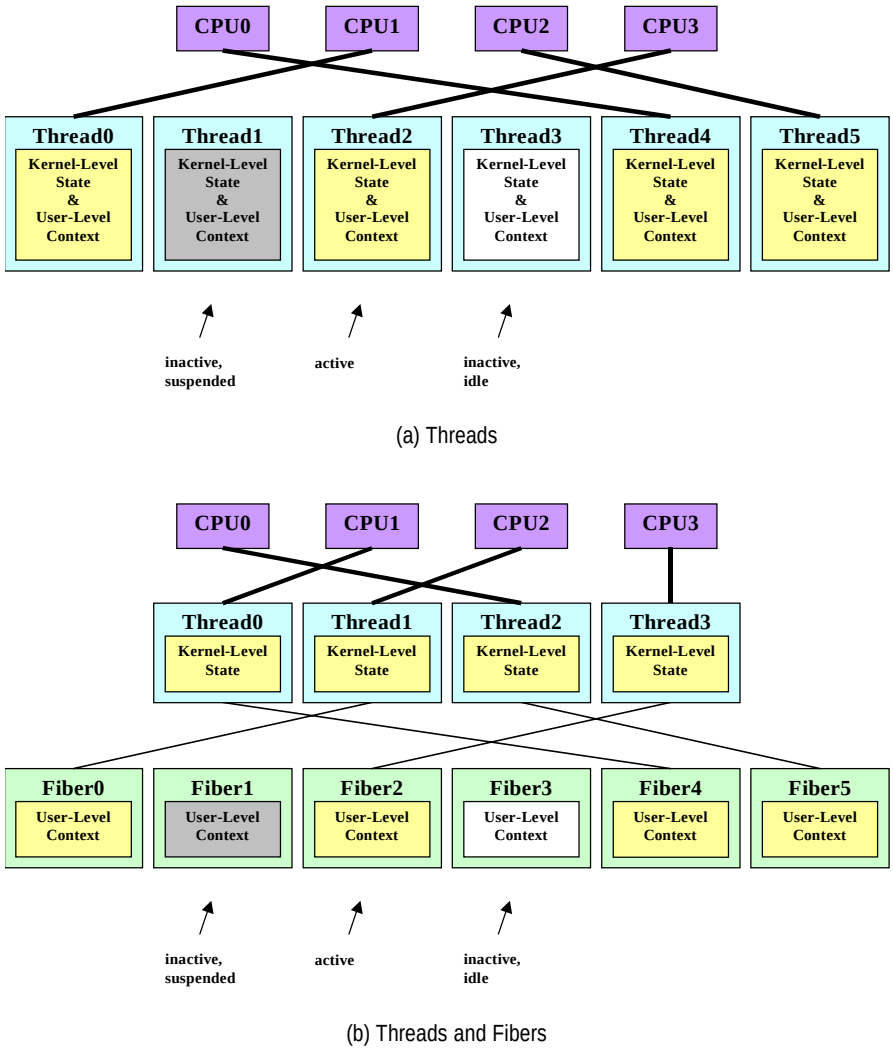


Figure 2: Relationship between threads and fibers

implementation of IML.

Threads or Tasks

In IML, tasks are represented by task descriptor blocks (TDB). A TDB contains a function pointer, a list of arguments to that function, and for parallel loops, a starting index, a dispatch counter, a minimum chunk size, a loop scheduling policy, and a pointer to a loop descriptor block (LDB). A LDB contains a completion counter and a pointer to the parent context of the loop. Each task is represented by a single TDB except for a parallel loop, which can be divided into one or more TDBs that share a single LDB.

Figure 3(a) is an example of a parallel loop, whose body is represented by the function `add_vectors_body`(). Figure 3(b) illustrates the relationships between the TDBs and the LDB for this parallel loop. The number of iterations `n` is assumed to have the value 1000. In this example, the parallel loop is divided into four TDBs of 250 iterations each (the initial value of the dispatch counter). The completion counter in the LDB is initialized to 1000, the number of iterations in the parallel loop. The parent context in the LDB is set to the address of the fiber executing the function call `iml_DOALL()`. The TDBs are enqueued into the task queues, where they wait to be scheduled for execution. Finally, the fiber executing `iml_DOALL()` yields to another fiber to participate in task execution.

Distributed Shared Queue (DSQ) and Load Balancing

In order to avoid contention on a centralized queue, IML uses multiple task queues distributed across multiple threads. Each thread owns a (local) queue, and can also access (remote) queues owned by other threads in order to achieve balanced load distribution. For systems with many processors, a hierarchical DSQ implementation may be preferable to a flat implementation. However, since the current target of IML is a four-way SMP Pentium Pro processor based system, IML employs a flat DSQ implementation. The current scheduler accesses remote queues in a round-robin fashion after the local queue becomes empty. This scheduling policy enhances cache locality of local queue accesses when threads continue to schedule tasks from their local queues.

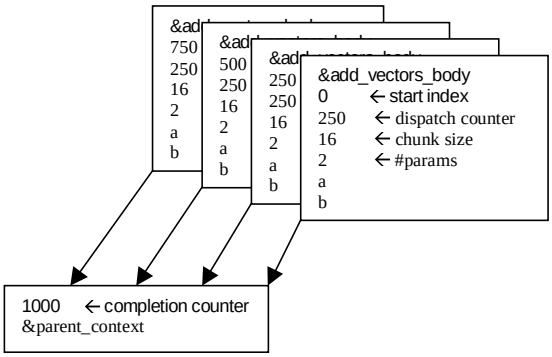
Dynamic load balancing is achieved when threads with empty local queues acquire tasks from remote queues. If a remote task is a non-loop task, the thread dequeues the task and executes it. If the task is a parallel loop, the thread splits the task in half [8], places one of the tasks in its local queue, and begins executing it. (This policy is chosen to maintain locality while reducing the cost of load balancing. User-specified minimum chunk size is honored in any loop-scheduling events.) Figure 4 illustrates the configuration of the DSQ. Threads and their local queues are connected by the bold lines. The regular lines represent the connections between threads and remote queues. IML allows users and external libraries to create multiple threads, and for each of these threads (multiple instances of Thread 0 in IML) to take advantage of IML. This enables rapid porting of existing threaded applications to IML.

Each task queue in IML is implemented as a stack to facilitate support for nested parallelism. When a thread encounters the first (outermost) level of parallelism, the newly created tasks are pushed onto the appropriate

```
void add_vectors(double *a, double *b, int n){
  int chunk = 16;
  int params = 2;
  iml_DOALL(&add_vectors_body, &n,
            &static_schedule, &chunk,
            &params, a, b);
  return;
}

void add_vectors_body(int *start, int *iters,
                      double *a, double *b) {
  int i;
  for(i = *start; i < (*start + *iters); i++){
    a[i] = a[i] + b[i];
  }
}
```

(a) A Parallel Loop



(b) TDB and LDB for (a)

Figure 3: A parallel loop, task descriptor blocks, and a loop descriptor block

stacks. Inner parallel tasks are pushed onto the local queue, hence increasing locality of task queue operations, as well as locality between inner parallel tasks.

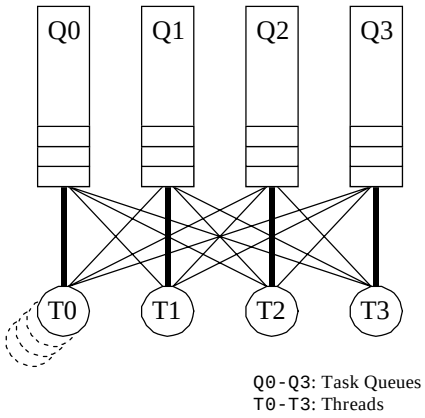


Figure 4: DSQ and threads

For example, in the case of doubly-nested parallel loops, the outer loop is distributed across multiple threads, while each inner loop is enqueued to the local task queue. Each thread continues executing iterations of the inner and outer loops from its local queue, until all the tasks in the local queue are completed. At this point, threads acquire tasks from remote queues making it possible for them to participate in the execution of inner parallel loops from other threads. By enqueueing all the inner loop iterations to the local queue, locality among these iterations is exploited.

Lock-Free Stack

The task queue stack is implemented without software locks [9] by using the iA instruction `lock CMPXCHG8B`, which performs an atomic compare and exchange operation. Figure 5(a) and (b) illustrate enqueue and dequeue operations of the pointer P2, respectively. In the enqueue operation (Figure 5(a)), the `lock CMPXCHG8B` instruction compares the pair "Empty-Top" (brown) against the stack top (green), and if the comparison succeeds, the stack top is replaced by the pair "Top-P2" (blue). When the comparison fails, the operation must be repeated with the new stack top. In the dequeue operation (Figure 5(b)), the top of the stack, which is the value to be dequeued, is used to construct the pair "Top-P2." The `lock CMPXCHG8B` instruction compares the pair "Top-P2" and the stack top, and replaces the stack top with "Empty-Top" if the comparison succeeds.

Unfortunately, the lock-free implementation allows only one access point to a task queue. Therefore, a thread obtains a remote task from the top of a remote task queue, even though outermost parallel tasks are found at the bottom of the task queue.

Process Stack Management

Exploitation of parallelism requires multiple execution contexts to be active simultaneously. In IML, each of

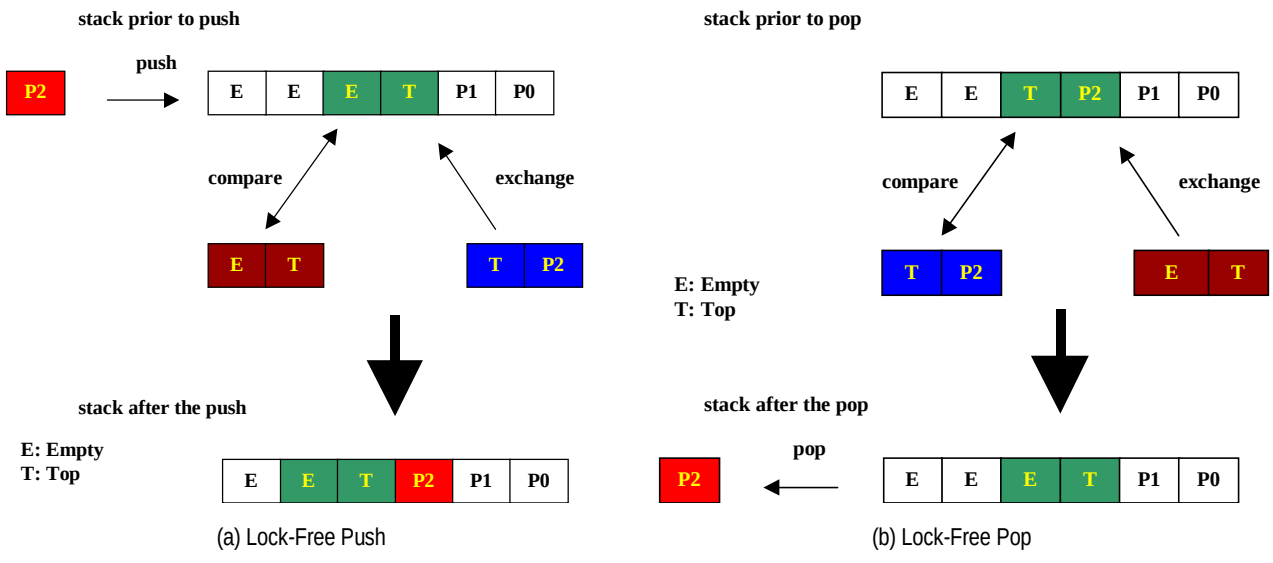


Figure 5: Lock-free push/pop operations

these contexts corresponds to an active Windows NT* fiber. The collection of stacks from active and suspended fibers resembles a cactus stack. Figure 6 illustrates the structure of the execution contexts for a parallel loop nested inside a COBEGIN section. The difference between this structure and a true cactus stack is that all the variables in parent contexts that are needed by child contexts are passed via parameters, instead of being accessed by a static linkage pointer.

In IML, when a new level of parallelism is invoked, the parent context is immediately suspended, and child contexts are initiated from a pre-allocated and recycled fiber pool. Upon completion of the parallel section, one of the active children contexts resumes the parent context [1].

Compiler-Generated Parallel Code

Parallel programming, compared to sequential programming, is a difficult and error prone process. Methods to automate or semi-automate this process are of great value to programmers. Automatic tools, such as automatic parallelizing compilers, are the ultimate tools that programmers can use to parallelize programs. Ideally, these compilers relieve the programmer of all the concerns of parallelization. However, two decades of research in parallel optimization has shown that optimal parallelization is often not achieved solely through automatic methods. In fact, semi-automatic parallelization techniques allow the user to guide the pre-processor or compiler in parallelizing the code. Fully and semi-automatic parallelizing methods are discussed in the following paragraphs.

Before discussing the two methods of parallelizing code, however, we first need to discuss how to transform code in order to interface with the IML. The transformation for parallel loops is described here, but a similar

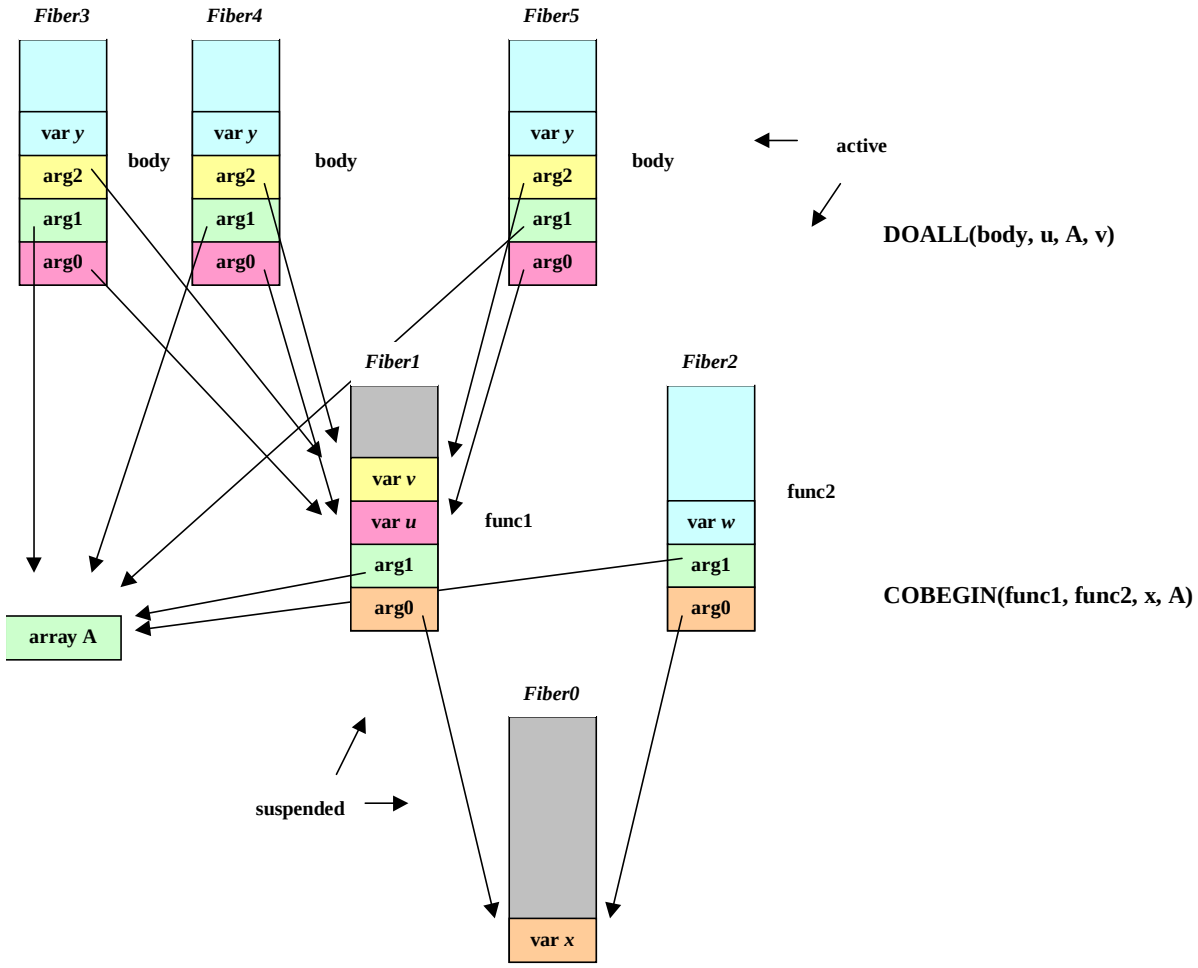


Figure 6: Stack configuration for a DOALL loop nested in a COBEGIN section

transformation is needed for parallel tasks. For each parallel loop, a new function is created that contains the loop body and the necessary support code. The shared and private variables of the loop are determined. Private variables (i.e., those with no cross-iteration dependencies) are redefined as local variables in the new function. The loop iteration variable is also declared as a local variable. All other variables are classified as shared and are declared as formal parameters of the new function. At the original site of the parallel loop, the body of the loop is replaced by a call to the IML entry point, `iml_DOALL`. All the information needed to execute the loop in parallel is passed to this entry point. This consists of the number of iterations, a pointer to the newly created function, the list of shared variables, the loop scheduling type, and the minimum chunk size. Some needed support code is also inserted around the call site.

Fully Automatic Parallelization

As mentioned above, the most convenient, but not necessarily the optimum, way to construct parallel programs is to utilize fully automatic tools such as parallelizing pre-processors or compilers that handle both the discovery of parallelism and the translation of parallel constructs. Two compilation systems, based on IML, were developed to facilitate fully automatic parallelism.

The Parafrase-2 parallelizing compiler, developed at the University of Illinois, was enhanced to output a transformed source code file with calls to the IML. The solid line path in the left side of Figure 7 shows this completely automatic path, which relinquishes the programmer from any parallelization effort. Parafrase-2 inserts all the necessary source code to manage the detected parallelism.

At Intel Microcomputer Research Labs (MRL), a parallel optimization module was added to the Intel compilers. This module accepts as input a standard intermediate representation of the source code produced by the front ends. Control and data flow analysis is performed on the intermediate form, and data dependence analysis is done on its loop constructs to discover loops with no loop carried dependencies (i.e., DOALL loops). These loops are then marked for a translator to convert them to the form required by the IML interface. This path is shown in solid lines in the right side of Figure 7.

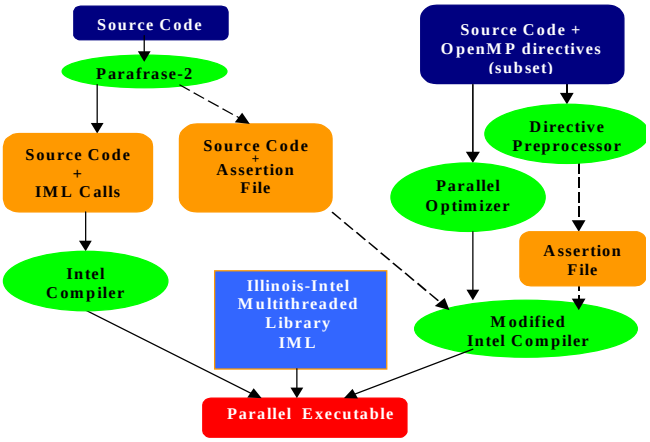


Figure 7: Parallel code generation

Semi-Automatic Parallelization

As stated earlier, fully automatic parallelization has its limitations. Fortunately, these limitations can be overcome by providing supplemental information about the program to the compiler or pre-processor, including information that cannot be determined at compile time. This information, which can be represented in many forms, such as directives, external assertion files, or interactive questioning by the compiler during compilation, is critical in the generation of efficient code.

Two semi-automatic methods have been implemented, corresponding to the diagonal path and the rightmost path (dashed lines) of Figure 7. Both methods encode parallel information in an assertion file, which the Intel C/C++ and FORTRAN compilers have been extended to access.

To automate the assertion file generation process, Parafrase-2 has been extended to generate an assertion file (along with a source code file). This process is fully automatic when Parafrase-2 generates efficient parallel code. However, when the code generated by Parafrase-2 does not perform adequately, the information in the assertion file can be augmented by the programmer to increase the performance of the parallel executable.

Another method is to encode parallelism in the source code via OpenMP directives explicitly. The augmented source code is then passed through a directive preprocessor, which generates an assertion file from the directives. The assertion file and the source code are then given to the modified Intel compilers to generate the parallel executable.

These semi-automatic methods detach the identification and the exploitation of parallelism. Parafrase-2 or the programmer identifies the parallelism in the program, while the modified Intel compilers transform the program to exploit this parallelism. Compared to the scheme where IML calls are inserted by Parafrase-2, the configurations using the assertion file increase the accuracy of the analysis performed by the modified Intel compilers.

Performance

Several experiments were performed to measure and evaluate the benefits of IML. These experiments were performed on a system with the following configuration:

- Intel System: Four 200MHz Pentium Pro processors, each with 256KB L2 cache; 4 way interleaved memory 512MB (60ns Fast Page Mode), Matrox MGA Millenium graphics card with 4MB VRAM
- Microsoft Windows NT* Server version 4.0
- Intel C/C++/FORTRAN Compiler version 2.4 (for compilation of IML and application programs)
- Microsoft Macro Assembler version 6.11d (for compilation of IML)
- Illinois-Intel Multithreading Library version 1.1

Intel System Memory Subsystem Performance

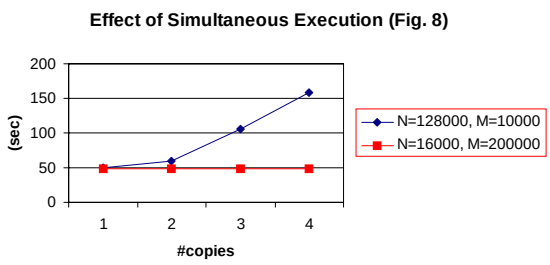
Before proceeding with the experiments that present the results with IML, a simple experiment was conducted to determine the impact of the memory subsystem performance on the results.

The effect of main memory bandwidth was evaluated using the code segment in Figure 8. When the array `block` fits into the L2 cache, almost perfect cache locality is achieved, resulting in very few main memory accesses. On the other hand, a large number of cache misses on the L2 cache were observed for larger sizes of the array `block`.

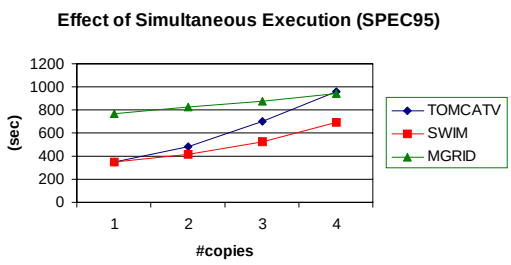
```
double a=0, block[N];
for(j=0;j<M;j++){
  for(i=0;i<N;i++){
    a += block[i];
  }
}
```

Figure 8: Code segment for memory subsystem test

Figure 9(a) illustrates the performance degradation of this code when multiple copies of this program are executed simultaneously. By running multiple independent processes of the same program, the experiment creates increased requirements on the bandwidth to main memory. For small sizes of array `block`, four copies of the program are executed without any performance degradation. When the size of array `block` is larger than the L2 cache (and thus each process now initiates more main memory accesses than the case of small array sizes), a performance degradation of approximately 220% is observed for four copies. This behavior is not limited to the test case. For example, three SPECfp95 benchmarks, MGRID, SWIM, and TOMCATV show 22%, 97%, and 174% slowdown, respectively, when four copies are concurrently executed (Figure 9(b)). The performance degradation observed in Figure 9 is attributable to the bandwidth between secondary cache and main memory. Therefore, it can be improved by performing cache locality optimizations. In the following experiments, no manual cache locality optimizations were performed.



(a) Execution Time of Test Code in Fig. 8



(b) Execution Time of SPECfp95 Codes

Figure 9: Effect of simultaneous execution

BLAS3

BLAS3 is a library package for matrix-vector operations. Several complex BLAS3 library functions from a preliminary version of the Intel Math Kernel Library (MKL) were ported to IML. MKL uses a conventional multiprocessing library, which exploits only a single level of parallelism. The computational kernels of these functions are written in FORTRAN and iA assembly with cache locality optimizations.

The speedup curve of one of these library functions, CGEMM, is presented in Figure 10 and can be seen to scale linearly. As the problem size increases, a moderate increase in the speedup is observed. The figure also illustrates that there is no significant difference in performance between IML and MKL. Unlike MKL which is a non-queue-based, singly nested, loop-only library and hence highly tuned, IML is a queue-based, runtime system that supports any mix of arbitrarily nested loop and functional threads, and hence is better suited for a larger class of application codes. Thus, one would expect that the additional functionality and general-purpose nature of IML would increase the overhead cost. Due to efficient implementation, this is not the case as is clear from Figure 10, and IML incurs thread management overhead comparable to fine-tuned libraries that support single loop only parallelism.

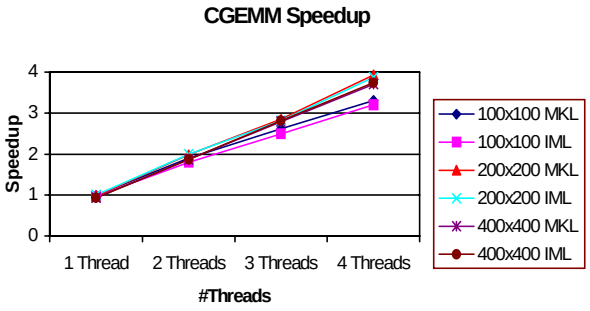


Figure 10: CGEMM speedup

SPECfp95

Three of the SPEC95 floating-point benchmarks, MGRID, SWIM, and TOMCATV were parallelized by the Parafrase-2 compiler. The benchmarks are numeric intensive and highly parallel. The solid lines of Figure 11 show the speedup curves for these benchmarks as measured on the actual system. Automatic parallelization with the Intel compiler produced similar results. As expected, the poor scaling is the result of the limited memory bandwidth. Extrapolating from these benchmark results and the performance of the memory subsystem for each benchmark from Figure 9(b), projected speedups, shown as the dotted lines in Figure 11, are obtained. These speedup curves correspond to a hypothetical system with sufficient main memory bandwidth.

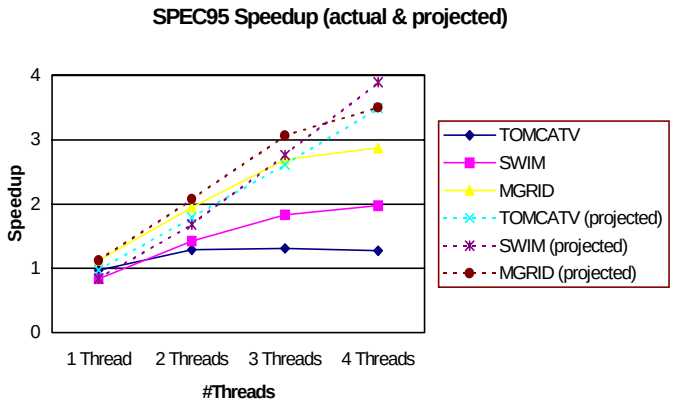
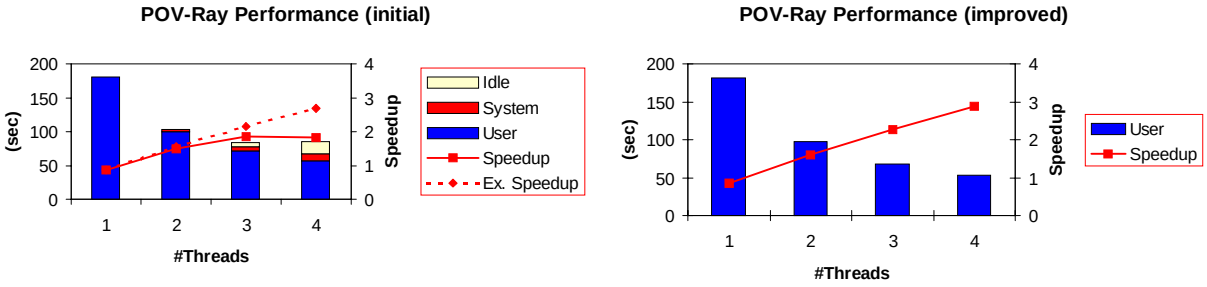


Figure 11: SPEC95 Speedup (actual & projected)

POV-Ray for Windows*

POV-Ray is a ray-tracing software package available to the public. This application is highly parallel since every pixel can be processed independently. In this experiment, however, only the parallelism between horizontal scan lines was exploited. The performance of the initial port is shown in Figure 12(a).



(a) Initial Performance

(b) Improved Performance

Figure 12: POV-Ray for Windows* performance

Execution times are represented by the bar graph. The colors blue, red, and white correspond to user, system, and idle time of the parallelized POV-Ray, respectively. The solid line represents the speedup over the original source distribution. The system and idle times, depicted in the bar graph, are due to the mutual exclusion inside `malloc ()`. The dashed line represents the projected speedup, computed only from the user time.

To eliminate this system level overhead, a second port enclosed the `malloc ()` function calls with user-level mutual exclusions, resulting in the performance shown in Figure 12(b). The system level overhead was eliminated, and linear speedup was obtained. Although the second port performs better than the initial port, it still suffers from serialization in the `malloc()` routine. A truly parallel implementation of `malloc()` would allow for even greater performance gains.

Conclusions

In this paper we have described IML, the Illinois-Intel Multithreading Library designed to support various types of parallelism efficiently. IML extends substantially the degree of available support for multithreading (found in other experimental or commercial systems) by providing the capability to express nested loop and cobegin/coend parallelism. Users can benefit from IML in terms of a reduction in development time by expressing parallelism in the IML API. To further assist the application developer, the Parafrase-2 compiler at the University of Illinois and the Intel FORTRAN Compiler have been modified to analyze programs to detect parallelism (automatically and with directives) and to generate calls to IML. Performance of automatically generated parallel code for SPECfp95 applications with IML is the same as hand-coded parallel programs.

Acknowledgments

This work was supported in part by a grant from Intel Corporation, in part by DARPA under grant MDA 904-96-C-1472, and in part by ONR under grant N00014-94-1-0234.

References

[1] Chow, J-H. and Harrison, L., Microtasking Recursive, Parallel Programs, *In Proc. of Int'l Conf. on Parallel Processing Vol. 2: Software*, 1990

[2] Girkar, M. Functional Parallelism: Theoretical Foundations and Implementation, *Ph.D. Thesis*, University of Illinois, 1992.

[3] Girkar, M. and Polychronopoulos C., Automatic Extraction of Functional Parallelism from Ordinary Programs, *IEEE Trans. on Parallel and Distributed Systems* Vol. 3, No. 2, 1992.

[4] Girkar, M. and Polychronopoulos, C. Extraction of Task-Level Parallelism, *ACM Trans. on Programming Languages and Systems* Vol. 17, No. 4, 1995.

[5] IML Home Page. http://www.csrd.uiuc.edu/IML.

[6] OpenMP Home Page. http://www.openmp.org.

[7] Polychronopoulos, C. *et.al.*, PARAFRASE-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors, *Int'l J. of High Speed Computing* Vol. 1, No. 1, pp. 45-72.

[8] E. D. Polychronopoulos, "Scheduling Heuristics for Multiprocessors," PhD Thesis in progress, Laboratory for High-Performance Computing, University of Patras, 1997.

[9]Valois, J. Implementing Lock-Free Queues, *Proc. of Int'l Conf. on Parallel and Distributed Computing Systems*, 1994, pp. 64-69.

[10] Win32 Fiber APIs. Microsoft Corporation.

Authors' Biographies

Milind Girkar received a B.Tech. from the Indian Institute of Technology, Mumbai, an M.Sc. from Vanderbilt University and a Ph.D. from the University of Illinois at Urbana-Champaign, all in computer science. Currently, he is a software engineer in Intel's Microcomputer Research Labs where he works on parallelizing compilers and Java Just-In-Time compilers. Before joining Intel, he worked on a compiler for the UltraSPARC platform at Sun Microsystems. His e-mail address is mgirkar@gomez.sc.intel.com.

Mohammad R. Haghighat is a software engineer at Intel's Microcomputer Research Labs where he works on parallelizing compilers and Java Just-In-Time compilers. He holds a B.Sc. in Computer Science and Engineering from Shiraz University, and an M.Sc. and a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. He is the author of a book on symbolic analysis for parallelizing compilers. His e-mail address is mhaghigh@gomez.sc.intel.com.

Paul Grey did his B.Sc. in Applied Physics at the University of the West Indies and his M.Sc. in Computer Engineering at the University of Southern California.

Currently he is a staff software engineer at Intel's Microcomputer Research Labs, researching compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., Sun Microsystems, and Silicon Graphics. His research interests include optimizing compilers, parallel computer architectures and 3D computer graphics. His e-mail address is pgrey@gomez.sc.intel.com

Hideki Saito received his B.E. degree in information science from Kyoto University in 1993. Currently, he is a Ph.D. candidate in the Department of Computer Science, and a research assistant in the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. At CSRD, he participates in the PROMIS parallelizing compiler project. His research interests include computer architectures and program optimizations for parallel processing. His e-mail address is saito@csrd.uiuc.edu.

Nicholas Stavrakos received his B.Sc. degree in computer engineering from the University of Illinois at Urbana-Champaign in 1994. Currently, he is a Ph.D. candidate in the Department of Electrical and Computer Engineering, and a research assistant in the Center for Supercomputing Research and Development at the University of Illinois. At CSRD, he participates in the Parafrase-2 and PROMIS parallelizing compiler projects. His research interests include parallelizing compilers, symbolic analysis, and multithreaded code generation. His e-mail address is stavrako@csrd.uiuc.edu.

Constantine D. Polychronopoulos is a Professor in the Department of Electrical and Computer Engineering and the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He received his Ph.D. from the University of Illinois at Urbana-Champaign in 1986, his M.Sc. from Vanderbilt University in 1982, and his B.Sc. from the University of Athens in 1980.

His research interests are on compilers and architectures for high-performance computer systems, multithreading, and multiprocessor operating systems. He has published extensively on parallelizing compilers and scheduling, and has been leading the Parafrase-2 and PROMIS projects at CSRD. He was the recipient of a 1989 NSF Presidential Young Investigator award, and is a Fulbright Scholar. Some of the results of his research team have been implemented in several commercial systems by DEC, Cray Research, Convex, Alliant, SGI and others. His research work has been funded by NSF, DARPA, ONR, and industry. His e-mail address is cdp@csrd.uiuc.edu.

Appendix: Examples for IML API

This appendix gives the examples for IML API functions. The code segments presented in this paper are simplified for explanatory purposes. Further details on the usage of IML can be found in the IML Home Page [5].

Basic API Functions

In this section, the usage of basic API functions is demonstrated using the original code shown in Fig. A-1. In this example, all three loops (i, j, and k) are parallel, and the two outermost loops (j and k) can be executed simultaneously.

```
double A[M][N], B(N), c;
for(j=0;j<M;j++){
  for(i=0;i<N;i++){
    A[i][j] = i * j;
  }
}
for(k=0;k<N;k++){
  B[k] = k;
}
c = foo(A, B, N, M);
```

Figure A-1: Code Segment for Fig. A-2 to A-5.

Converting the j- and k-loops to DOALL results in the code shown in Fig. A-2. The k-loop is executed after the j-loop is completed.

```
    double A[M][N], B(N), c;

    iml_DOALL(&jloop, &M,
              &static_schedule,
              &chunk, &params,
              A, N);
    iml_DOALL(&kloop, &N,
              &simple_schedule,
              &chunk, &params, B);
    c = foo(A, B, N, M);



  void jloop(int *start,
             int *iters,
             double *A, int *N){
    for(j=*start;
        j<*start+*iters;j++){
      for(i=0;i<N;i++){
        A[i][j] = i * j;
      }
    }
  }
  void kloop(int *start,
             int *iters,
             double *B){
    for(k=*start;
        k<*start+*iters;k++){
        B[k] = k;
      }
    }
  }
```

Figure A-2: Using `iml_DOALL()` for Outer Loops

The i-loop can also be converted to DOALL as in Fig. A-3. Unlike conventional libraries that would internally execute the i-loop described in this fashion as a sequential loop, IML can actually execute it in parallel.

```
  double A[M][N], B(N), c;

  iml_DOALL(&jloop, &M,
            &static_schedule,
            &chunk, &params, A, N);
  iml_DOALL(&kloop, &N,
            &simple_schedule,
            &chunk, &params, B);
  c = foo(A, B, N, M);



void jloop(int *start,  int *iters,
           double *A, int N){
  for(j=*start;
      j<*start+*iters;j++){
    iml_DOALL(&iloop, &N,
              &self_schedule, A, j);
  }
}
void kloop(int *start, int *iters,
           double *B){
  for(k=*start;
      k<*start+*iters;k++){
    B[k] = k;
  }
}
void iloop(int *start, int *iters,
           double *A, int j){
  for(i=*start;
      i<*start+*iters;i++){
    A[i][j] = i * j;
  }
}
```

Figure A-3: Using `iml_DOALL()` for all loops

Furthermore, the j- and k-loops can be executed simultaneously, using `iml_COBEGIN()` (Fig. A-4) or `iml_EnQ()` (Fig. A-5).

Extended API Functions

The usage of the extended API functions is demonstrated using the original code shown in Fig. A-6. The reduction operation can be performed in parallel, where each thread reduces to its own variable, and global reduction across the result of the thread-wise reduction is performed afterwards (Fig A-7). If a sequential section of the program persists for a period of time, the programmer or the compiler can use `iml_ReInitMultiThread()` to reduce the number of active threads (Fig. A-8).

```
  double A[M][N], B(N), c;

  iml_COBEGIN(&tasks, &jloop_0,
              &kloop_0,
              &params, A, B, N, M);
  c = foo(A, B, N, M);


void jloop_0(double *A, double *B,
             int N, int M){
  iml_DOALL(&jloop, &M,
            &static_schedule,
            &chunk, &params, A, N);
}
void kloop_0(double *A, double *B,
             int N, int M){
  iml_DOALL(&kloop, &N,
            &simple_schedule,
            &chunk, &params, B);
}
void jloop(int *start, int *iters,
           double *A, int N){
  for(j=*start;
      j<*start+*iters;j++){
    iml_DOALL(&iloop, &N,
              &self-schedule, A, j);

  }
}
void kloop(int *start, int *iters,
           double *B){
  for(k=*start;
      k<*start+*iters;k++){
    B[k] = k;
  }
}
void iloop(int *start, int *iters,
           double *A, int j){
  for(i=*start;
      i<*start+*iters;i++){
    A[i][j] = i * j;
  }
}
```

Figure A-4: Using `iml_COBEGIN()`

```
  double A[M][N], B(N), c;

  iml_EnQ(&jloop_0, A, B,
          N, M, &cnt, &c);
  iml_EnQ(&kloop_0, A, B,
          N, M, &cnt, &c);



void jloop_0(double *A, double *B,
             int N, int M,
             int *cnt, double *c){
  iml_DOALL(&jloop, &M,
            &static_schedule,
            &chunk, &params, A, N);
  if (iml_DecAndFetch(cnt)==0){
    iml_EnQ(&foo_0, A, B, N, M, c);
  }
}
void kloop_0(double *A, double *B,
             int N, int M,
             int *cnt, double *c){
  iml_DOALL(&kloop, &N,
            &simple_schedule,
            &chunk, &params, B);
  if (iml_DecAndFetch(cnt)==0){
    iml_enQ(&foo_0, A, B, N, M, c);
  }
}
void foo_0(double *A, double *B,
           int N, int M, double *c){
    *c = foo(A, B, N, M);
}
void jloop(int *start, int *iters,
           double *A, int N){
  for(j=*start;
      j<*start+*iters;j++){
    iml_DOALL(&iloop, &N,
              &self-schedule, A, j);

  }
}
void kloop(int *start, int *iters,
           double *B){
  for(k=*start;
      k<*start+*iters;k++){
    B[k] = k;
  }
}
void iloop(int *start, int *iters,
           double *A, int j){
  for(i=*start;
      i<*start+*iters;i++){
    A[i][j] = i * j;
  }
}
```

Figure A-5: Using `iml_EnQ()`

```
double a, B(N);
for(i=0;i<N;i++){
  a += B{i];
}
```

Figure A-6: Code Segment for Fig. A-7 and A-8

```
  double a, A(NCPU), B(N);

  iml_DOALL(&iloop, &N,
            &static_scheduling,
            &chunk, &params,
            A, B);
  for(i=0;i<NCPU;i++){
    a += A[i];
  }


void iloop(int *start, int *iters,
           double *A, double *B){
  ID = iml_GetThreadID();
  for(i=*start;
      i<*start+*iters;i++){
    A[ID] += B{i];
  }
}
```

Figure A-7: Using `iml_GetThreadID()`.

```
  double a, A(NCPU), B(N);

  iml_DOALL(&iloop, &N
            &static_scheduling,
            &chunk, &params,
            A, B);
  iml_ReInitMultiThread(1);
  // suspend all other threads
  for(i=0;i<NCPU;i++){
    a += A[i];
  }


void iloop(int *start, int *iters,
           double *A, double *B){
  ID = iml_GetThreadID();
  for(i=*start;
      i<*start+*iters;i++){
    A[ID] += B{i];
  }
}
```

Figure A-8: Using `iml_ReInitMultiThread()`.