Using a Functional Language And Graph Reduction To Program Multiprocessor Machines Or Functional Control of Imperative Programs

Lal George[1]
Gary Lindstrom[2]

UUCS-91-020

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

October 9, 1991

Abstract

This paper describes an effective means for programming shared memory multiprocessors whereby a set of sequential activities are *linked* together for execution in parallel. The *glue* for this linkage is provided by a functional language implemented via graph reduction and demand evaluation. The full power of functional programming is used to obtain succinct, high level specifications of parallel computations. The imperative procedures that constitute the sequential activities facilitate efficient utilization of individual processing elements, while the mechanisms inherent in graph reduction synchronize and schedule these activities.

The main contributions of this paper are: 1) an evaluation of the performance implications of parallel graph reduction; 2) a demonstration that the mechanisms of graph reduction can obtain multiprocessor performance uniformly surpassing the best uniprocessor implementation of sequential algorithms running on a single node of the same machine, and 3) an illustration of our method used to program a real world fluid flow simulation problem.

Keywords: Functional programming, lazy evaluation, graph reduction, Standard ML, type checking, parallel languages.

[1] Supported in part by the National Science Foundation under Grant CCR-8704778 and an IBM Fellowship Award.

[2] Supported in part by the National Science Foundation under Grants ASC-9016131 and CCR-8920971.

1 Introduction

The benefits of using a functional language to program multiprocessor machines have long been heralded [18]; however, such claims remain largely unsubstantiated in practicality. We investigate the factors posing the principal challenges in the use of a functional language to program multiprocessor machines. Our particular domain of interest is the solution of large engineering problems such as seismic wave simulations, weather prediction and fluid flow problems.

There are several strategies for evaluating functional programs. We will adopt *normal order* evaluation since graph reduction, its most widely used implementation technique, molds well on a parallel machine. Parallelism under normal order is obtained from (i) the parallel evaluation of arguments to strict operators such as $+, -, \times, ...$, and (ii) optionally, the speculative evaluation of expressions whose value may eventually be required. Together, normal order and speculative evaluation elegantly encompass a wide variety of multiprocessing effects such as the overlapped generation and consumption of data.

Functional languages evaluated by graph reduction offer the promise of high level control of multiprocessors. A programmer is freed from the direct orchestration of parallel evaluation, and may succinctly express complex asynchronous algorithms through the use of powerful abstractions such as higher order functions and polymorphic types. This is in contrast to languages based on low level primitives such as channels, or parallel control constructs like fork-join or forall.

However, fulfillment of the promises of graph reduction has not been immediately forthcoming. It is difficult for graph reduction to compete in speed with traditional methods for programming von Neumann machines, which form the processing units of today's multiprocessor architectures. The performance of parallel graph reduction is limited by the fine granularity of tasks that seems inherent in such systems when implemented in their purest forms. Further, the side-effect free nature of functional languages usually entails unacceptable copying and recycling overheads on store management.

2 Experimental Setting

2.1 Language

We have developed a programming system called TML (tam-Il) for Tiny ML. This may be thought of as a functional subset of SML [10] evaluated using normal order reduction. The syntax is identical to that of SML, with the addition of one construct - the familiar spark [2], used to spawn the *speculative* evaluation of expressions. The spark construct is similar in syntax to a let declaration, e.g.:

```
spark val x = e1 val y = e2 in e3 end
```

This expression spawns the speculative evaluation of **e1** and **e2**, binding their results to x and y, respectively. The value returned is that of **e3**, which may contain x and y as free variables.

2.2 Multiprocessor

All of our experiments were conducted on the BBN GP1000 Butterfly - a MC68020 based non-uniform access, shared memory multiprocessor running an early version of the MACH operating system. Each node has 4 Mbytes of local memory and is connected to other nodes by a delta switching network. With no other traffic on the switch a remote reference is five times slower than a local reference. Each node runs at the maximum clock rate of 16MHz. The nodes do not possess data caches.

2.3 Compilation

Our compilation scheme is based on the G-machine [11], which is generally regarded as one of the more efficient implementations of normal order evaluation. We have extended this compilation scheme to allow for efficient parallel evaluation using strictness information. The details of the compilation method have been previously reported [6, 15, 7], and will be omitted here. We will merely highlight some pertinent properties: i) a task is represented at runtime by a pointer to a graph to be reduced; ii) when the machine is heavily loaded parallel tasks are evaluated in line thus mimicking sequential execution; iii) if a task blocks a continuation is built and attached to the graph responsible for providing the awaited value, permitting the processor executing the blocked task to be reassigned to other work, and iv) strictness information is used to avoid context switching by pre-evaluating arguments that are guaranteed to be required. This increases the average basic block size and dramatically improves speed.

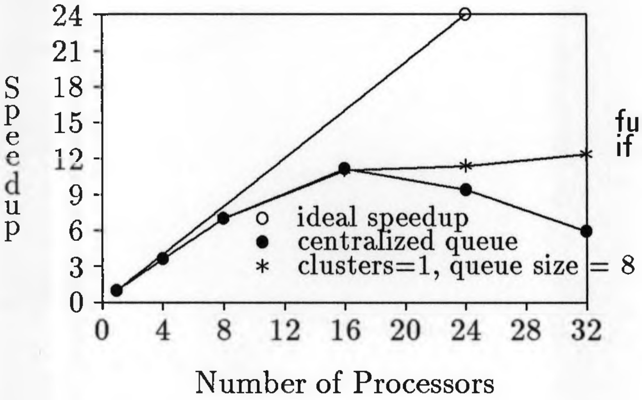
2.4 Runtime Support

It is important in any parallel processing implementation to provide high quality support for scheduling, memory allocation and garbage collection. Poor implementations of support facilities can dramatically affect performance and radically distort speedup effects. A novel scheduling structure was developed [8] for this system, along with a conventional *concurrent* stop and copy garbage collector. All quantitative measures reported in later sections include garbage collection time. For debugging purposes the runtime system was conditionally linked to the X11 window system library, and provided a graphical display of the activities on each processor.

In summary, we believe that we have implemented an efficient parallel graph reduction system.

3 Parallel Graph Reduction in Practice

We investigated the performance of several functional programs. In most cases the speed in comparison to the best sequential programs (usually written in C or SML) was poor. It will be difficult to persuade a scientist or engineer to recode his or her applications in a functional language only to be rewarded by a marginal speedup or even a substantial loss in performance. The reasons for this (and possible remedies) are dramatically illustrated by the fib1 (Fibonacci) function shown in Figure 1[3].



```
fun fib1 n =
if n < 2 then 1
     else spark val f1 = fib1(n-1)
          in f1 + fib1(n-2)
          end
```

Processors	1	4	8	16	24	32
centralized queue (sec)	99.1	27.5	14.19	8.9	10.55	16.8
cluster=1, queue=8 (sec)	-	-	-	9.05	8.72	8.03
Avg. tasks dequeued	1	2023	3344	5693	4346	4521
ATG (*sec/task*)	99.1	1.4e-2	4.2e-3	1.5e-3	1.7e-3	1.8e-3
Blocked computations	0	568	705	998	627	689

Figure 1: Performance of `fib1 25` function application

The function was written using the **spark** construct as shown in order to generate parallel tasks. Having carefully designed our compilation strategy and runtime system, we expected this computation to perform well. We conjectured that, after the initial flurry of parallel activity, the execution on each processor would default to in line execution and execute sequentially. This is clearly not the case as seen from the extremely large number of tasks dequeued, i.e. 2000-6000. For each of these tasks, scheduling and memory allocation overheads are incurred which directly impacted the average task granularity (ATG in units of *seconds/task*) of the program. Furthermore, the number of times a computation

[3] The figure show the performance of the centralized task queue structure and a distributed task queue structure [8].

is blocked and a continuation built was also high — 500-1000. This exacerbated context switching overhead. While the speedup was quite impressive, the maximum speed attained came nowhere near the sequential speed of SML or C. This example may be contrived but its characteristics are representative of a larger set of programs which we analyzed with similar results.

We draw the following insights from our experience in using parallel graph reduction (only some of which are illustrated by the particular example of the Fibonacci function):

1. Functional programming is an excellent vehicle for expressing parallel algorithms. Being relieved of the tedium of expressing communication, synchronization and scheduling constraints is a major benefit.

2. Parallel functional programs can easily be composed to produce larger parallel programs. Higher order functions and polymorphism are convenient for defining general parallel processing abstractions that can be reused in a variety of situations.

3. One pays a severe price for using graph reduction. Basic blocks are usually quite small, since primitive operators (e.g. $+$, $-$, $\times$ ...) must systematically check the evaluation status of their operands.

4. Graph reduction as a model of evaluation is not well matched to the basic processing elements on most multiprocessors, which are von Neumann machines rather than specialized or dedicated graph reduction processors.

5. The functional style can be quite clumsy when expressing solutions that involve *opportunistic* communication as found in parallel branch-and-bound algorithms, or manipulate large data structures such as arrays.

6. For good parallel performance, merely specifying the parallel computation is not enough. Granularity considerations are crucial.

In the subsequent sections we attempt to ameliorate these difficulties by restricting the use of graph reduction to what it does best: parallel task coordination.

4 Increasing the Granularity

By rewriting the fib1 function so that no parallelism is generated if the task granularity is sufficiently small, we notice a dramatic improvement in performance as illustrated in Figure 2. There are improvements in speed, the number of tasks dequeued, the number of blocked computations and the average task granularity. Speedup is sustained for well over 24 processors.

We can go one step further by performing the sequential tasks using applicative order evaluation, which is better suited to von Neumann processors. That is to say, the sequential tasks are executed as ordinary (call-by-value) ML functions. This suggests a model in which a functional language evaluated using graph reduction serves as the glue combining a set of sequential computations. The mechanisms of graph reduction are used to initiate tasks (demand values), notify recipients of results and handle the underlying concerns of parallel evaluation. We call the sequential tasks *functional processes.* They are *bona fide* functions taking inputs and producing answers — i.e., there are no side effects.

We carried this idea through to an implementation in which the functional processes were coded in C. This has the added benefit of making meaningful timing comparisons with sequential algorithms written in C. The main idea is illustrated using a matrix multiplication program where each row of the result matrix is computed in parallel. The core of the functional specification is:

```
fun matmult A B n =  let val Bt = transpose B n
                     in pmap (compute_row A Bt n) (from 0 (n-1))
                     end
fun pmap f [ ] = [ ]
  | pmap f (x::xs) =  spark val fx = f x  in fx :: pmap f xs  end
fun from n m =  if n>m  then [ ]  else from (n+1) m
```

The function compute_row (of arity 4) computes a row of the result matrix C (= A × B) of dimension n × n, the last argument being the row to compute. The function pmap forks off parallel activity. Note:

- The task granularity is that of a single row. It is trivial to extend this to two or more rows thereby adjusting the granularity of the computation.

- Although not evident in the program provided above, the function compute_row is executed imperatively in C. Its definition should be obvious.

- During execution, the application compute_row A B n is represented in the graph in the same manner as any other function application node, except that its code pointer invokes imperative code.

- We assume that all functional processes such as compute_row are strict in all their arguments, so the call to compute_row is only made when all its arguments are fully evaluated. The necessary control (scheduling, blocking, resumption etc.) required in the generation of these arguments is provided via the basic mechanisms of graph reduction. This is essential, since the functional processes cannot reasonably be designed to deal with delayed values.

- If each matrix were allocated on a single memory module then this implementation could perform very badly due to memory contention. The absence of processor data caches on our machine, the BBN GP1000, causes nonlocal memory references to result in accesses across the communication switch. For good performance, a local copy of the row being accessed should be made. This illustrates the fact that the programmer can utilize his or her knowledge of the machine (e.g. no data cache) and the algorithm (e.g. frequent access to certain structures) for better performance in specific cases.

Using the code exactly as written above (with the addition that each functional process made copies of the rows involved in the computation) we obtained the performance shown in Figure 3. We compared this with the sequential execution of a matrix multiplication routine written exclusively in C. The latter was not subject to the overheads of parallel processing support, and used the same algorithm as the parallel program. This convincingly *outperforms* both the GNU gcc and Green Hill C compilers executing the uniprocessor version of matrix multiplication, starting with 2 processors and obtaining 3 times the speed with 8 processors.

5 Controlling Imperative Programs

Using the above model we were able to program several interesting applications which in each case convincingly outperforms the gcc C language compiler. We conjecture that a large majority of actual parallel applications can be programmed under the model described. This is extremely beneficial, since parallel programs in this model are easily composable and amenable to proofs of correctness. In addition, portability is enhanced: (i) the functional specification acts as the machine independent part of the parallel program, and (ii) the functional processes are no harder to port to multiprocessor machines than ordinary sequential programs, which, indeed, is exactly what they are.

However, there remain certain applications that are troublesome in this model, such as the parallel manipulation of a single array data structure in seismic wave simulations or weather prediction programs. Existing functional languages fail to address such applications adequately, due to the latter's reliance on repeated parallel updates of a shared data structure.

We extend our computation model once more to allow functional processes to modify their inputs, resulting in *imperative processes*. The burden of respecting data dependence is assumed by the programmer. Most imperative languages like the IBM 3090 Parallel Fortran provide a **parbegin/parend** or **fork/join** construct that lets a programmer or automatic parallelizer ensure desired evaluation orders. We can provide a similar facility by defining the function forkjoin of two arguments f and g. These arguments are functions, where f () is forked on a different processor and g () is performed asynchronously.[4]

[4] The parentheses represent an argument of type unit in SML.

```
fun forkjoin(f, g) =  spark val f_proc = f ()
                      in g () ; f_proc ; ()
                      end
```

The join operation is performed by the sequencing operator[5] that ensures the necessary barrier synchronization of the two processes. We have expressed a large number of parallel constructs used in imperative languages including the fairly complex version of the **fork/join** construct described by Almasi and Gottlieb [1].

Using the idea behind the **fork/join** construct we can now write a quicksort program that modifies a single array as:

```
fun qsort A m n =
    if m >= n   then A
    else  let val mid = partition A m n
        in
            spark val q1 = qsort A (mid + 1) n
            in (qsort A m (mid - 1); q1; A)
            end
        end
```

The function partition is implemented as an *imperative process*. It takes an array and index range and partitions the array in place. Using a slightly modified version of qsort and sorting a 10,000 element array, the parallel execution was just barely able to surpass the speed of the sequential C compiler. Although quicksort may not be the best way to sort a list of elements in a functional language, the demonstration is an important one. Given an existing algorithm and associated data structures with potential for parallelism, it is essential to be able to realize that potential by evolution into a parallel program. Better results can reasonably be expected on problems such as weather prediction where the granularity is significantly larger.

6 TML in Practice

In practice, respecting data dependencies does not seem to be a difficult issue. Like SML, one is encouraged to program in a largely functional style, and data dependencies, where inescapable, are localized. When constructing a parallel algorithm one has a good idea of the computations that can be performed in parallel. These are exactly the *functional* or *imperative processes*. The power of functional programming is used to specify the

[5] Note that: e1 ; e2 ≡ case e1 of _ => e2

scheduling, synchronization and notification involved in the execution of these parallel tasks, which we feel is the most error prone aspect. This is also a popular approach in other languages where the algorithm design revolves around *data decomposition* and *partitioning* of data across the machine [22, 19].

The functional or imperative processes are viewed as black boxes that are called upon to perform a specific task. Hence the programmer has a clean separation and a clear view of how various parallel programs are composed and used. In this manner it is possible to evolve existing sequential code to run on a parallel machine.

7 A Real Demonstration

7.1 CML

We are extending the ideas reported in the previous paragraphs to the production quality Standard ML of New Jersey compiler. We have implemented our lazy functional framework in SML using the concurrency extensions provided by CML [20, 21]. CML provides two asynchronous communication primitives called transmit and receive that return a value of type α **event**. These represent a promise to perform the communication requested. The primitive sync of type (α **event** $\rightarrow \alpha$) is used to perform the actual synchronization. For example, accepting a message is equivalent to:

```
sync o receive
```

Using these dictions, it is possible to represent futures and delayed computations. In particular, the following module implements our lazy evaluation primitives:

```
signature LAZY =
  sig
    type 'a lazy
    val apply : ('a - > '2b) - > 'a - > '2b lazy
    val delay : ('a - > '2b) - > 'a - > '2b lazy
    val spark : ('a - > '2b) - > 'a - > '2b lazy
    val force : 'a lazy - > 'a
    val strict : 'a lazy - > 'a
    val seq : ('a lazy * 'b lazy) - > 'b
  end
```

The module LAZY specifies a type 'a lazy which represents a potentially unevaluated object. The construct for spark is no longer a let-like construct; rather, is a function of two arguments f and x. The spark function spawns the application of the function f to x, returning a lazy value. It is easy to guess the purpose of the other functions from their types and names, except to remark that **strict** is identical to **force** and **apply** is identical to **delay**.

In this manner it is possible to implement an environment of lazy operators, part of which is shown below:

```
signature LAZYINT =
 sig
   infix 7 | ∗ | ..
   infix 4 | < | ..
   ..
   val op | ∗ | : int lazy * int lazy − > int
   val op | < | : int lazy * int lazy − > bool
 end
signature LAZYLISTS =
 sig
   datatype 'a Llist = NIL | CONS of 'a lazy * 'a Llist lazy
   exception Hd
   exception Tl
   val hd : 'a Llist − > 'a
   val tl : 'a Llist − > 'a Llist
   val cons : 'a lazy * 'a Llist lazy − > 'a Llist
   val printLlist: ('a − > string) * 'a Llist − > unit
 end
```

Operators over numbers are surrounded by |, such as | + |, | ∗ |, ... With this framework one can define whatever data structures are of particular interest, e.g. lazy trees:

```
datatype 'a tree = LEAF of 'a lazy | TREE of 'a tree lazy * 'a lazy * 'a tree lazy
```

A major benefit of this approach is that the strong typing of SML ensures that all the **force** or **demand** operations are placed where required. Therefore, it is not possible to use a value of type **int lazy** where an value of type **int** is required. This approach is similar to that pursued by Mishra and Kuo [14] in relating strictness analysis to type inference. The burdensome tagging and untagging of data between the lazy functional language and the functional processes is now handled by the data destructuring of SML. Garbage collection is no longer a perplexing issue as the model does not use two distinct languages. This was a major concern in some of our earlier experiments.

7.2 SIMPLE

We have coded the SIMPLE [4, 3] benchmark (a fluid flow simulation program) using the concurrent model described above. A brief outline will be sketched here. The state of the simulation is a set of arrays representing physical properties like the velocity, pressure, and viscosity of the fluid in a sphere. These are used in a loop to generate the next set of arrays for the next time step. Each of the functions that generate these arrays is fairly heavy weight and requires embodies considerable computation.

There are several ways to implement this benchmark in our model. The main loop of one such approach is shown below:

```
fun runit () =
   let fun iter (i,state) =  if i = 0  then strict state
                             else iter (i-1, spark compute_next_state state)
   in iter (step_count, apply compute_initial_state ())
   end
```

Note that because the application of compute_next_state to state is evaluated speculatively, we get the effect of several iterations being executed together. This is similar to software pipelining, except that we are not pipelining instructions but rather coarse grained functions. The state itself is represented as a set of lazy objects that may be in any state of evaluation. The function that computes the next state is shown below:

```
fun compute_next_state state =
   let
      val (v,x,alpha,s,rho,p,q,epsilon,theta,deltat,c) = strict state
      val v'  = spark make_velocity (v, x, p, q, alpha, rho, deltat)
      val x'  = spark make_position  (x, deltat, v')
      ...
   in
     (v',x',alpha',s',rho',p',q',  epsilon',theta',deltat',c')
   end
```

Each component of the new state is created using speculative evaluation, manifested by calls such as spark make_velocity(...). The result of compute_next_state is a set of arrays where each array may be partially evaluated. Most of the functions like make_velocity use **fork-join** parallelism while generating their results. This is done in a fashion similar to that illustrated in the quicksort example. The application strict state merits some explanation. The function strict performs only one level of evaluation, and the call returns a set (or more precisely a tuple) of arrays that may be partially evaluated. This is essential for the pipelining effect of several iterations.

An implementation of CML on a Silicon Graphics multiprocessor is in progress and we expect to be able to report performance results in a final paper on this work.

8 Relation to Other Work

These ideas originated from a classical paper by Kahn on the semantics of a simple language for parallel processing [12]. This framework was later demonstrated in the language POP-2 by Kahn and MacQueen [13]. The language was emulated using coroutines and no performance results were reported.

Lucassen and Gifford describe a language FX for parallel computers that uses an effect system to discover expression scheduling constraints [16, 9]. If an expression performs no side effect operations to the store then it can be executed in parallel with other such expressions. Furthermore, if two expressions perform side effects to two different stores, then they too can be done in parallel. In our view, this sort of parallelism is limited. It excludes a large class of parallel algorithms that manipulate large arrays where concurrent threads operate on an independent segment of the same array. Some index range analysis in some form is required to make this parallelism possible which is beyond the scope of their current type system.

A bilingual form of evaluation exists in the language Strand [5] based on logic programming. The distinguishing feature of our approach is that it is based on a functional language where the full power of higher order functions can be used.

Our work is very similar to Delirium [17]. Here a programmer must learn two languages - Delirium which is a functional language and another language like C that serves as the main computing elements of the parallel program. The main drawback with this arrangement (which we discovered from our own experiments) is that the communication of data between the two languages becomes problematic. Indeed Delirium restricts this to be integers or arrays.

9 Future Work

As mentioned in §7.1, the incorporation of these ideas into the Standard ML of New Jersey are already under way. The major goals are:

- Parallel processing runtime support facilities in ML. Some work has already been done in the context of SML Threads.
- Implementation of graph reduction facilities as basic primitives in the compiler.
- For the present we will be satisfied with surrounding expressions with appropriate force and delay functions. However, there as been much work in the area of type theory on automatic coercions. We feel that these results can be applied to automate this process and are carrying such out an investigation.
- We hope to experiment with the Splash benchmarks [23], which must first be recoded in ML.

10 Conclusions

This paper describes an effective means of programming shared memory multiprocessors by linking together a set of sequential activities to be executed in parallel. The *glue* for these sequential activities is provided by a functional language implemented using graph reduction and demand evaluation. The full power of functional programming is used to succinctly specify parallel computations at a high level. Imperative procedures that constitute the sequential activities make it possible to effectively program the individual processing elements of the machine. The mechanisms inherent in graph reduction are reserved for the production of arguments and the notification of results. To a degree the resulting parallel programs are easily composable and are amenable to proofs of correctness which are becoming increasingly important at a time when programs must withstand rigorous analysis, particularly for safety critical operations.

We have outlined a new implementation under development in Standard ML of New Jersey and demonstrated how the strong typing can be used to ensure the correct interaction between the lazy functional language and the functional or imperative processes.

References

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989. ISBN 0-8053-0177-1.

[2] Chris Clack and S. L. Peyton Jones. The four-stroke reduction engine. In *Conf. on Lisp and Functional Programming*, pages 220–232. ACM, August 1986.

[3] W.P. Crowley, C.P. Hendrickson, and T.E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[4] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, July 1987. (Simultaneously published as IBM/T.J. Watson Research Center Research Report 12686).

[5] I. Foster and S. Taylor. *Strand, New Concepts in Parallel Programming.* Prentice Hall, 1990.

[6] L. George. Efficient normal order evaluation through strictness information. Master's thesis, University of Utah, March 1987.

[7] L. George. An abstract machine for parallel graph reduction. In David MacQueen, editor, *Proc. Symposium on Functional Languages and Computer Architecture*, pages 214–229, London, September 11-13, 1989. Springer-Verlag.

[8] L. George. A scheduling strategy for nonuniform access shared memory multiprocessors. In *Proc. of International Conference on Parallel Processing.* IEEE Computer Society, August 1990.

[9] D.K. Gifford, P. Jouvelot, J.M. Lucassen, and M.A. Sheldon. *FX-87 Reference Manual.* MIT, Laboratory for Computer Science, 1.0 edition, 1987. Tech. Report Number MIT/LCS/TR-407.

[10] R. Harper, R. Milner, and M. Tofte. The definition of Standard ML. Technical Report ECS-LFCS-88-62, Dept. of Computer Science, Univ. of Edinburgh, Aug. 1988. Version 2.

[11] T. Johnsson. Efficient compilation of lazy evaluation. In *Proc. Symp. on Compiler Construction*, Montreal, 1984. ACM SIGPLAN.

[12] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing 74*, pages 471–475, 1974. North-Holland.

[13] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. *Information Processing 77*, pages 993–998, 1977. North-Holland.

[14] Tsung-Min Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*, pages 260–272. ACM, September 1989.

[15] G. Lindstrom, L. George, and D. Yeh. Generating efficient code from strictness annotations. In *TAPSOFT'87: Proc. Second International Joint Conference on Theory and Practice of Software Development*, pages 140–154, Pisa, Italy, March 1987. Springer Lecture Notes in Computer Science No. 250.

[16] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, Jan 1988.

[17] S. Lucco. Parallel programming with coordinating structures. In *Conf. Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–208. ACM, January 1991.

[18] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

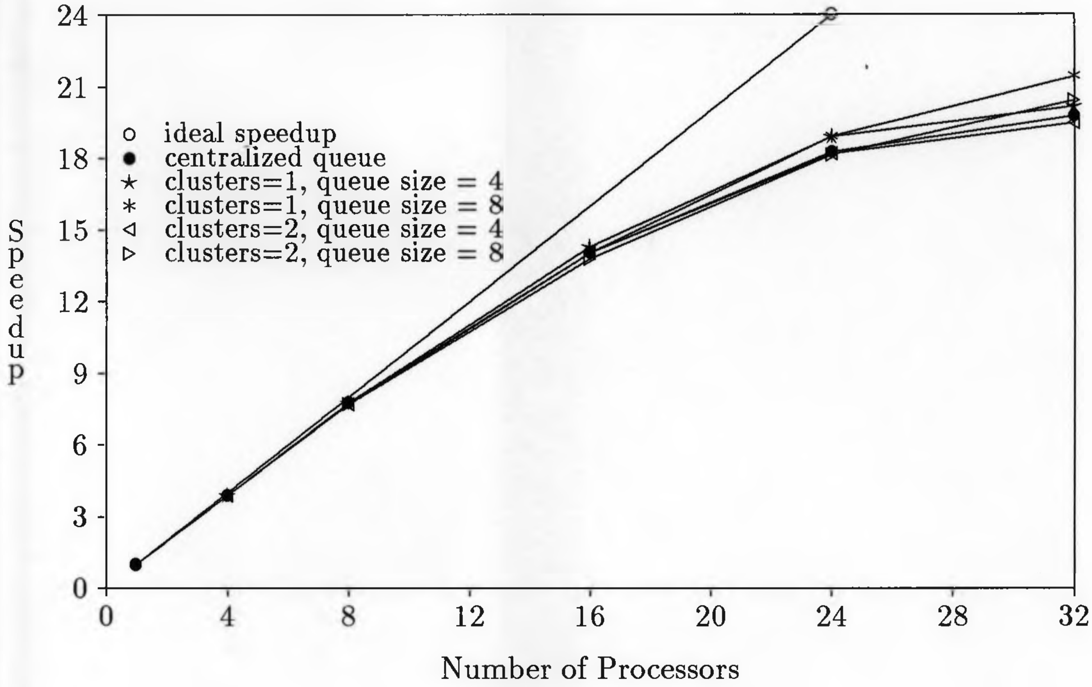
[19] K. Pingali and A. Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. Technical Report TR 90-1084, Cornell University, January 1990.

[20] J.H Reppy. First-class synchronous operations in Standard ML. In *Proc. of the SIGPLAN'88 Conf. on Prog. Language Design and Implementation*, pages 250–259. ACM, June 1988.

[21] John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305. ACM, June 1991.

[22] A. Rodgers and K. Pingali. Process decomposition through locality of reference. In *Proc. of the 1989 ACM SIGPLAN Symposium on Programming Language Design and Implementation*. ACM, June 1989.

[23] Jaswinder P. Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.



Processors	1	4	8	16	24	32
centralized queue (sec)	80.0	20.6	10.3	5.7	4.38	4.05
cluster=1, queue=8 (sec)	-	-	-	5.7	4.23	3.74
cluster=2, queue=8 (sec)	-	-	-	5.8	4.41	3.92
Avg. tasks dequeued	1	161	178	318	437	555
ATG (*sec/task*)	80.0	1.2e-1	5.8e-2	1.7e-2	1.0e-2	7.7e-2
Blocked computations	0	95	111	198	288	324

Figure 2: Performance of `fib2 25` function application

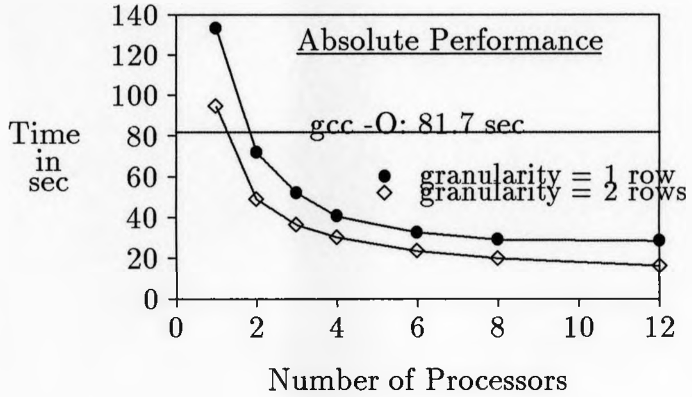


Figure 3: Performance of matrix multiplication on 256×256