EQUALS – The Next Generation

Owen Kaser[*]
Dept. of MSCS
UNB at Saint John
Saint John, NB, Canada.
owen@unbsj.ca

C.R. Ramakrishnan[†]
Dept. of Comp. Science
SUNY at Stony Brook
Stony Brook, NY 11794, USA.
cram@cs.sunysb.edu

R.C. Sekar [‡]
Bellcore
445 South St.
Morristown, NJ 07962, USA.
sekar@bellcore.com

Abstract

EQUALS is a system for parallel evaluation of lazy functional programs implemented on a Sequent Symmetry. A preliminary implementation of EQUALS [4] was used to establish the validity of Normal Form (NF) demand propagation and memory reclamation via reference counting. However, that implementation did not exploit *vertical parallelism*, where the arguments to a function can be evaluated in parallel with the function itself. The overheads of vertical parallelism can be as high as the overheads in systems that do not propagate NF demand. Hence careful integration of vertical parallelism with horizontal parallelism (parallelism among the arguments to a function) is needed to fully benefit from NF-demand propagation. In this paper we describe schemes to harness vertical parallelism within the context of the current EQUALS system. We identify the aspects of the runtime system (task management) that affect the overall efficiency of the combined system. We also provide preliminary performance figures indicating the effectiveness of the selected schemes.

1 Introduction

Functional languages offer a conceptually simple approach to programming parallel computers. Detection and exploitation of parallelism in functional programs is simpler than in imperative programs due to referential transparency. The parallelism inherent in a functional program is classified into:

1. *Horizontal Parallelism*, or parallelism *among* the arguments of a function. In such cases, the needed arguments of a function can be evaluated in parallel before the function is invoked.
2. *Vertical Parallelism* (also known as *stream* parallelism) which arises when the arguments of a function can be evaluated in parallel with the function itself.

Whereas programs in strict languages exhibit only horizontal parallelism, lazy languages express horizontal as well as vertical parallelism.

EQUALS is a system for parallel execution of lazy functional programs, implemented on the Sequent Symmetry. A distinguishing feature of EQUALS over previous implementations on shared memory machines (e.g., Buckwheat [3], GAML [6], and the $\langle \nu, G \rangle$-machine [1]) is the propagation of exhaustive Normal Form (NF) demand in addition to the traditional Weak Head Normal Form (WHNF) demand. A preliminary implementation of EQUALS [4] showed that NF-demand propagation leads to low parallel overheads. However, that implementation did not exploit vertical parallelism that arises naturally in many programs. If naively implemented, the overheads due to vertical parallelism can be as high as that in systems propagating WHNF demand alone. Hence careful integration of vertical and horizontal parallelism is needed to maximize the benefits due to NF-demand propagation. In this paper we describe schemes to harness, *i.e.*, to exploit and control vertical parallelism within the context of the current EQUALS system.

The paper is organized as follows. We present an overview of the EQUALS system in the next section, and describe the scheme for exploiting horizontal parallelism. In section 3 we motivate the use of vertical parallelism, develop a scheme to exploit it, and discuss issues in controlling parallelism. In section 4 we

[*]Supported by NSERC grant OGP0155967.
[†]Supported in part by the NSF (CCR-9102159, CDA-9303181 and INT-9314412).
[‡]This work was performed while R. C. Sekar was at Stony Brook and supported in part by NSF CCR-9102159.

describe the features of the runtime system along with proposed modifications needed to efficiently evaluate programs with vertical parallelism. We describe the current state of the implementation in section 5, and identify future work.

2 Overview of EQUALS

The EQUALS system consists of a compiler and a runtime support system. The compiler uses *ee*-strictness analysis developed in [8] to detect parallelism and translates the source program to C via a combinator-based intermediate language. The compiled code is executed under the control of a runtime system that provides facilities for task creation, switching and synchronization. The runtime system also manages the resources, such as memory, needed to execute a task.

The goal of compiled code is to normalize a given input expression. The EQUALS compiler generates two versions of code for each function, which are invoked under WHNF and NF demand respectively. Partial applications (closures), delayed evaluations (thunks) and head normal forms are all represented uniformly as graphs (also referred to as terms or expressions) in the heap. Once evaluated, a graph is overwritten with its normal form, thus sharing the normal-form computation.

If multiple subexpressions need to be evaluated in order to normalize an input expression, these subexpressions can be evaluated in parallel. Subexpressions that are evaluated in parallel are taken up by individual tasks, which execute the compiled code on their private stacks. Since there may potentially be many more parallel tasks than the resources available (e.g., processors, shared memory), tasks are created only when they are deemed useful. Some of these decisions are made at compile time itself; for instance, the compiler will never emit code to create a new task, unless it can generate code for work to be done concurrently by the existing task. At runtime, on the other hand, opportunities for additional parallelism will be passed up, if sufficient parallel resources are not available.

Although these tasks might be executed as UNIX processes, it is too expensive to do so. Instead, EQUALS implements a mechanism for managing lightweight tasks, where tasks are executed under the control of evaluator processes (one per processor). All runnable tasks are placed in a global ready queue, which is used to share and balance the system load. Given an input expression, a task to normalize this expression is created and placed in the ready queue. Whenever idle, the evaluator processes pick up tasks from the ready queue and start their execution. If a task needs the value of a subexpression that is currently being evaluated by another task, then the first task is suspended, awaiting the subexpression's evaluation. The evaluator then begins execution of a task from the ready queue. Once an expression's evaluation is complete, all tasks awaiting its evaluation are released to the queue.

2.1 Intermediate Language

We represent the code generated for a functional program using an abstract intermediate language, described in [4]. The constructs of the language relevant to this paper are those used to build and dismantle graphs, evaluate expressions locally and in parallel, and synchronization barriers, described below.

Build *var* = *expr*: Build (without evaluating) the graph corresponding to *expr* and store it in *var*.

GetChild *n* **of** *t* **in** *var*: Store the *n*-th child of the graph in *t* in *var*.

Eval *var* = *expr* **to** *extent*: Evaluate the expression *expr* to *extent* in the current task, storing its value in *var*; *extent* may be NF (default) or WHNF.

RemoteEval *var* = *expr* **to** *extent*: Evaluate the expression in *expr* to *extent* in parallel with the current task. When the evaluation is complete, store its value in *var*.

WaitFor *extent* **of** *var*: Suspend current task until the expression corresponding to *var* is evaluated to *extent*.

2.2 Compiled Code

Consider a fragment of the *n-Queens* program shown in in Figure 1. The figure shows a rule that composes multiple solutions (*add_columns*) using the function *append*, and the rules that define *append*. The code corresponding to the *add_columns* rule under NF demand is given in Figure 2, and the code generated for *append* (under NF demand) appears in Figure 3. Note that *append* is strict in both its arguments under NF demand, *i.e.*, whenever normal form computation of t_1 or t_2 diverge, the normal form computation of $append(t_1, t_2)$ also diverges. Hence its arguments are evaluated to NF, potentially in parallel, in the code for *append_columns* before *append_NF*

```
...
add_columns(x, n, bd:bds) =
    append(add_col(x, n, bd),
        add_columns(x, n, bds)
...

append(x:xs, ys) = x:append(xs, ys)
append(nil, ys) = ys
```

Figure 1: Fragment of *N-Queens* program.

```
fun add_columns_NF(t1, t2, t3)
    case root(t3) of
        ...
        cons:
            GetChild 1 of t3 in t4
            GetChild 2 of t3 in t5
            RemoteEval t6 = add_col_NF(t1, t2, t4)
            Eval t7 = add_columns_NF(t1, t2, t5)
            WaitFor NF of t6
            Eval t8 = append_NF(t6, t7)
            Return t8
        ...
```

Figure 2: Fragment of code generated for *add_columns*.

is invoked. Since whenever *append_NF* is called its arguments are already in NF, the return value of *append_NF* is always in NF.

2.3 Runtime Structures

The memory contains two main areas, namely, heap and stack. The heap is divided into equal-sized *nodes* that are used to construct the graphs. The value fields of a node hold the symbol of the node and pointers to its children. The status fields of a node include NF bit that indicates that the graph rooted at this node is in normal form; and InProc bit that is set when the graph rooted at this node is under evaluation.

A task consists of a graph to reduce and a stack to use in the reduction. Instead of keeping stack frames in heap, stacks are built out of blocks of contiguous memory, obtained from stack space. The stack of a task is a linked list of such blocks. When the task overflows the current block, a new block is allocated and linked to the current stack; on underflow, the block is returned to the free pool.

See [5] for a detailed description of the structures used in the runtime system.

```
fun append_NF(t1, t2)
    case root(t1) of
        cons:
            GetChild 1 of t1 in t3
            GetChild 2 of t1 in t4
            Eval t5 = append_NF(t4, t2)
            Build t6 = cons(t3, t5)
            Return t6
        nil:
            Return t2
```

Figure 3: Code generated for *append*.

3 Vertical Parallelism

In the example in Figure 1, although whenever $append(t_1, t_2)$ is evaluated to NF, both t_1 and t_2 need to be in NF, *append* can compute portions of its output even before t_2 is examined. Moreover, only the spine of t_1 is directly inspected by *append*. Hence *append* can be invoked *before* the normal form computations of t_1 and t_2 are completed. Thus, *append* can consume its inputs *incrementally*. Furthermore, portions of the normal form output of *append* can be produced without inspecting any argument in full. The incrementality in the input-output behavior of *append* resembles an element in a pipeline[1]. This implicit (vertical) parallelism present in *append* was not exploited by the scheme sketched in the previous section.

3.1 Exploiting Vertical Parallelism

Note that in presence of vertical parallelism, the arguments of a function are not guaranteed to be evaluated to any desired extent when the function is invoked. Hence the code that computes the normal form of a function must test the extent of evaluation of an argument before inspecting any part of the argument. In case the required argument is not yet evaluated to the necessary extent, the current function's evaluation must await till the argument evaluation is complete. The code for *append* that incrementally consumes its input is given in Figure 4.

Observe that *append_NF_vert* consumes its inputs incrementally, but does not produce incremental out-

[1] Hence, vertical parallelism is also called *pipeline* parallelism, or producer-consumer parallelism.

```
fun append_NF_vert(t1, t2)
    WaitFor HNF of t1
    case root(t1) of
        cons:
            GetChild 1 of t1 in t3
            GetChild 2 of t1 in t4
            Eval t5 = append_NF_vert(t4, t2)
            Build t6 = cons(t3, t5)
            WaitFor NF of t3
            Return t6
        nil:
            WaitFor NF of t2
            Return t2
```

Figure 4: Vertical Parallel code (naive) generated for *append.*

put. Hence *append_NF_vert* can participate as the final consumer in a pipeline, but cannot form the middle of any pipeline. Note that when the second rule of *append* is applicable we can output the *cons* node that forms the head of its normal form *immediately.*

To achieve this, firstly, we need to de-link the data and control flow in the compiled code. A function, instead of simply **Return**-ing its output, writes the output at a pre-assigned location, which is passed to the function as an additional argument. This location is also passed to any task that depends on the output of the function.

Secondly, we need a mechanism to mark the (as yet) unevaluated parts of a graph. We can use closures that represent partially evaluated functions for this purpose. A closure not only *marks* a graph as unevaluated, but also specifies how to build this graph, and is expensive to build. Note that since some task is actively engaged in completing the evaluation, we need to only mark the unevaluated part as "under construction"; the specifics of how to build this graph are irrelevant. This is done using a special nullary function symbol, that is overwritten with the result when that part of the graph is (eventually) evaluated. This overwriting operation is analogous to the normal rewriting operation on the graph. Tasks are suspended pending overwriting of this symbol and resumed in the usual manner, as described in section 2.

The code generated using the above scheme for *add_columns* and *append* appear in Figures 5 and 6 respectively. In the figures, the instruction **Makebot** x creates a graph node (pointed to by x) and marks it as "under construction"[2]; the instruction **OverWrite** $d = s$ overwrites the node in d with the graph in s and releases the tasks awaiting the evaluation of graph d; the instruction **SetNF** x sets the **NF** bit of node x.

```
fun add_columns_NF_Vert(t1, t2, t3, ret)
    case root(t3) of
        ...
        cons:
            GetChild 1 of t3 in t4
            GetChild 2 of t3 in t5
            MakeBot t6
            RemoteEval add_col_NF_Vert(t1, t2, t4, t6)
            MakeBot t7
            RemoteEval add_columns_NF_Vert(t1, t2, t5, t7)
            Eval append_NF_Vert(t6, t7, ret)
        ...
```

Figure 5: Fragment of Vertical Parallel code generated for *add_columns.*

```
fun append_NF_Vert(t1, t2, ret)
    WaitFor HNF of t1
    case root(t1) of
        cons:
            MakeBot t3
            GetChild 1 of t1 in t4
            Build t5 = cons(t4, t3)
            OverWrite ret = t5
            GetChild 2 of t1 in t6
            Eval append_NF_vert(t6, t2, t3)
            WaitFor NF of t4
            SetNF ret
        nil:
            WaitFor NF of t2
            OverWrite ret = t2
```

Figure 6: Vertical Parallel code generated for *append.*

3.2 Controlling Vertical Parallelism

Opportunities for parallel evaluation must be passed up whenever the cost of exploiting the parallelism outweighs the benefits. This situation arises when either the available parallelism exceeds the system resources, or the synchronization costs are comparable to the computation cost itself.

Note that, in presence of vertical parallelism, instead of waiting once before invocation of a function for evaluation of an argument, we await evaluation of different parts of the argument as needed. The

[2]Note that this can be done by simply turning the InProc bit on.

additional parallelism offered by the pipelining of argument and function evaluations must be sufficient to overcome the additional synchronization costs. Note that the synchronization in presence of vertical parallelism corresponds to synchronization at WHNF level and hence the cost can be as high as that in systems without NF-demand propagation.

Horizontal parallelism is relatively inexpensive to exploit compared to vertical parallelism due to lower synchronization costs. Hence, in a system that exploits both types of parallelism, vertical parallelism must be avoided whenever sufficient horizontal parallelism is available. In a system that exploits only horizontal parallelism, we avoid creating new parallel tasks when the system load is high. Firstly, we extend this scheme in presence of vertical parallelism. The threshold at which parallelism is forsaken differs for horizontal and vertical tasks – creation of horizontal tasks are throttled at a higher load than the creation of vertical tasks. Secondly, a vertical task remains incremental only when there is insufficient horizontal load on the system. All vertical tasks cease incremental output when the system's horizontal work load exceeds a threshold. Thus creation of horizontal tasks is given a priority in the system over creation and execution of vertical tasks.

In addition to the higher synchronization costs, vertical parallel schemes create many more tasks than horizontal schemes. Most of these tasks are suspended awaiting output of other tasks and the system load, measured as the size of ready queue, does not reflect this. Consider the fragment of *n-Queens* program in Figure 1. In that example, *add_col* is a producer and *append* is a consumer in a pipeline. If the producer is faster than the consumer, there are no bottlenecks in the pipeline. However, *append* consumes its argument rapidly enough to often await (involving a task switch) the output of *add_col*. In the current implementation, the total number of tasks (ready, active and suspended) in the system is also used to throttle the creation of vertical tasks. However, this is a global scheme that controls the overall parallelism in the system instead of avoiding extension of the pipelines with bottlenecks.

We are investigating two approaches to identifying useful parallelism – *compile time schemes* that transform a program so as to avoid fine-grained synchronization; this involves analysis methods to estimate task granularity (*e.g.*, see [2]), and *run-time schemes* which avoid parallelism in pipelines containing tasks that frequently switch between the ready and wait queues.

4 Runtime Issues

To efficiently harness useful vertical parallelism, it is necessary to minimize the task synchronization overheads. To this end, in the current runtime system (see [5]) the inactive tasks (either freshly created or suspended) and the evaluator tasks are uniformly represented using a *suspension*. A suspension consists of a task control block pointing to the top of stack, and the stack containing volatile registers, base pointer and code pointer as its top frame. A task is resumed by *entering* its suspension that consists of simply restoring registers and the base pointer and `return`-ing to the code.

When a task is suspended, the context switches to the evaluator, which picks the next available task from the ready queue. Since the ready queue is a shared resource, its access needs to be carefully designed to avoid bottlenecks. The *split-lock* queue used in EQUALS shows good base performance and scalability – enqueue and dequeue take only $7\mu s$ and $5\mu s$ respectively on 2 processors, and $38\mu s$ and $22\mu s$ on 10 processors in the worst case. Task creation overheads are also low. For instance, creating a new parallel task and enqueuing it takes only $48\mu s$. Creating a new parallel task that returns immediately, and then awaiting it requires $125\mu s$ if an idle processor is available. Otherwise, the parent needs to be suspended and reinvoked, and $150\mu s$ is typical.

The number of tasks created in presence of vertical parallelism also stresses the memory management. Apart from the large number of suspended tasks, the scheme to create vertical tasks as sketched in Section 3 creates a large number of tasks that await NF of an argument, and once resumed, set the return graph to NF and end. The function $append\_NF\_Vert$ (Figure 6) is a good example of such a task. In many cases, it turns out that the return graph is no longer of interest to any task, and is freed immediately after the NF bit is set! Hence both the graph and the stack associated with the task are held much longer than they need be. Note that we can avoid most of the above operations based on the reference counts that are already maintained on the nodes. We are currently implementing a task cleanup mechanism tuned to vertical tasks, to reclaim stacks whenever the return graph is no longer needed.

Based on the current implementation of the runtime system, the *n-Queens* program (on a 10×10 board) shows a speedup of 13.2 on 20 processors, despite creating numerous tasks (over 41,000) and allocating over 520,000 nodes.

5 Conclusion

In this paper we described a scheme to integrate vertical parallelism with horizontal parallelism in the presence of NF-demand propagation, and mechanisms to efficiently support this scheme. The focus of our current effort is in identification of *useful* vertical parallelism: designing compile-time schemes to avoid fine-grained synchronization, and run-time schemes to avoid extending pipelines showing little usable parallelism. Implementation of a task cleanup mechanism for early release of the resources associated with a vertical task is underway. Although preliminary results are encouraging, much work is needed in evaluating the execution mechanisms over larger programs.

References

[1] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle\nu, G\rangle$ machine. In *Functional Programming Languages and Computer Architecture*, 1989.

[2] S. Debray, N. Lin and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *ACM Conference on Programming Languages Design and Implementation*, 1990.

[3] B. Goldberg. Buckwheat: Graph reduction on shared-memory multiprocessor. In *ACM Conference on Lisp and Functional Programming*, 1988.

[4] O. Kaser, S. Pawagi, C.R. Ramakrishnan, I.V. Ramakrishnan and R.C. Sekar. Fast Parallel Implementation of Lazy Languages – The EQUALS Experience. In *ACM Conference on Lisp and Functional Programming*, 1992.

[5] O. Kaser, C.R. Ramakrishnan and R.C. Sekar. A High Performance Runtime System for Parallel Evaluation of Lazy Languages. In *Intl. Symposium on Parallel Symbolic Computation*, 1994.

[6] L. Maranget. GAML: A parallel implementation of lazy ML. In *Functional Programming Languages and Computer Architecture*, 1991.

[7] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 1992.

[8] R.C. Sekar, S. Pawagi and I.V. Ramakrishnan. Small domains spell fast strictness analysis. In *ACM Symposium on Principles of Programming Languages*, 1990.