Management Of Heterogeneous Parallelism On Shared Memory Multiprocessors†

Athar B. Tayyab    Jon G. Kuhl

Dept. of Electrical and Computer Engineering
and
Center for Computer Aided Design
University of Iowa, Iowa City, IA 52242

ABSTRACT
This paper considers the problem of management of heterogeneous parallelism on shared memory parallel processing systems. Heterogeneous parallelism includes all forms of inter-instruction parallelism. This may include both explicitly coded and compiler generated forms. It is argued that support mechanisms are needed to efficiently manage heterogeneous subcomputations at run-time, and that the important issues in the design and implementation of these mechanisms are different from those previously studied for support of loop-level parallelism.

An empirical study of an actual application is presented. This study confirms that the specific nature of run-time support mechanisms can dramatically impact the performance of a parallel program. The use of simple, syntactically closed constructs is suggested.

I. INTRODUCTION
A large amount of parallel processing research has focused on techniques for manually or automatically formulating computational problems in a manner such that they exhibit the potential for parallel execution. A less studied, but equally important, problem involves the relationship between a parallel computation and the underlying mechanisms used to map its computational threads onto a parallel machine. The most commonly studied form of program parallelism has been the parallel execution of loop iterations [8,11,13,19]. In the recent work of Beckmann and Polychronopoulos [4], the relative performance of various scheduling and synchronization policies for iterations of a DOALL loop structure were compared. The overheads associated with scheduling of loop iterations onto processors and synchronizing the computation were modeled analytically and the efficiency of various scheduling and synchronization policies were compared as a function of the granularity and variance of loop iterations times.

In addition to loop-level parallelism, programs may exhibit the potential for more heterogeneous forms of parallelism, e.g. at the subroutine-level, or at the level of groups of instructions. Active research is now being carried out in developing restructuring compilers that will detect heterogeneous parallelism in a program and translate the program into parallel form [12,14,17], compiler directives that allow users to express program parallelism explicitly [1], and other tools and packages that can exploit common sources of program parallelism. However these techniques either assume compile-time construction of static computational threads, or do not consider the problem of execution time mapping at all. For reasons to be discussed later, we do not believe that compile-time thread building is adequate. Instead some form of execution time scheduling and control mechanisms may be required.

The techniques used for managing loop-level parallelism cannot be directly applied to the general problem of heterogeneous parallelism for several reasons, including the following:
(i) The heterogeneous nature of a set of parallel subcomputations may violate the common assumption of identically distributed execution times typically applied to loop iterations in the analysis of loop-level parallelism.
(ii) The generally more constrained data dependencies among heterogeneous subcomputations of a program (as opposed to iterations of a DOALL loop) limit the ability to arbitrarily adjust task granularities (intervals between synchronization points), as is commonly done in comparing scheduling techniques for loops.
(iii) The structure of a heterogeneous parallel application, as defined by data dependencies, may be more general than that imposed by closed constructs such as nested DOALL and DOACROSS loops.

Little work has been reported relative to analysis of the effectiveness of various scheduling and synchronization schemes for supporting heterogeneous parallelism. However, these issues may be extremely important, especially with regard to successfully exploiting relatively fine-grained heterogeneous parallelism, such as might be identified by a parallelizing compiler.

This paper discusses the issues involved in implementing support mechanisms for heterogeneous parallelism for shared memory multiprocessor systems.

The paper also presents the results of an empirical study involving a real application. The application exhibits a large amount of subroutine-level parallelism, with data dependencies that severely constrain the ability to adjust task granularities. The study allows us to compare the performance of different management mechanisms for this application. Although the studied application is characterized by explicitly identified subroutine-level parallelism, the general conclusions can be applied to other forms of heterogeneous parallelism, including compiler generated forms.

II. TASK MANAGEMENT ISSUES FOR SHARED MEMORY MIMD SYSTEMS

In this section, we will discuss efficiency issues related to task management operations for shared memory multiprocessor systems. These issues will be discussed in the

† This work was partially supported by the National Science Foundation under grant number ECD-8715307
context of managing a set of interrelated parallel subcomputations. A natural abstraction of heterogeneous parallelism is the tasking model. A task, for the purpose of this paper, is defined as a non-preemptive unit of computation defined by an independently executable subcomputation which may require local context and may have parameters. This definition is general enough to allow natural program structures such as procedures and subroutines. A task can be scheduled for execution by another task and it can schedule other task(s) for execution. Such operational precedence constraints as well as inter-task data dependencies can be represented as a directed graph with no (redundant) transitive edges. Such a graph is referred to as a task graph.

Once the parallelism in an application code is expressed as a task graph, parallel processing support mechanisms may be needed to schedule and synchronize tasks on a particular multiprocessor system such that the data dependencies are satisfied. Serial bottlenecks and overheads in these mechanisms can reduce the parallel performance of the application. Techniques must be employed to eliminate these bottlenecks and reduce the associated overheads to exploit the maximum amount of parallelism available. In general, the smaller the granularity of parallelism allowed, the more parallelism can be detected in programs. A task management scheme with large overhead will nullify the performance gained from a large amount of parallelism with small program pieces.

Compile-time construction of completely static computational threads is sometimes used to minimize scheduling overheads [14]. This approach, however, has a limited scope and is often unacceptable for several reasons. First, it fixes the number of processors on which the compiled application can run. In addition, poor balancing of computation load among processors may result. Furthermore, the conditional control flow of the program may be too complex to allow static threads to be easily specified. Finally, the approach does not entirely avoid overheads anyway, since barriers or similar synchronization mechanisms must be used to enforce dependencies among threads.

As an alternative, specific task management mechanisms can be inserted into the application by a programmer or a parallelizing compiler to schedule and synchronize tasks in accordance with identified inter-task dependencies. Overheads associated with such mechanisms need to be examined carefully. In any task-based parallel processing support environment, there are four major sources of overheads: (i) task creation, (ii) task scheduling, (iii) task startup, and (iv) task termination.

There are overheads associated with creating a task. For example, if the task needs local context, a piece of stack may have to be allocated for the task being created. Memory allocation is a costly operation as it involves operating system calls. As pointed out in [18], a better approach is to create all the tasks during compilation. This approach requires that all the tasks executed at run-time be defined at compile-time.

Placing a new task into the scheduling pool also introduces overheads, due mainly to serialization of operations on the scheduling pool. Fully parallel queues were proposed in [9]. Such schemes can reduce scheduling overheads but require sophisticated hardware support. Parallel linked lists for processor scheduling are described in [15]. The scheme, however, relies on mutual exclusion to resolve contention and consequently serial bottlenecks can arise. Another approach would be to have several scheduling pools, one for each independent group of tasks. This would reduce the serial bottlenecks associated with a single, global scheduling pool but load balancing becomes difficult.

Scheduling $n$ child tasks of a task would involve processing overhead proportional to $n$. This approach is clearly unacceptable for large values of $n$. Preferred functionality is constant-time spawning of multiple tasks. One possible approach is to pre-build the scheduling queue for each independent group of tasks. Then the run-time operation will be reduced to dispatching this queue. Thus a group of tasks can be launched via a single, constant-time call.

To startup a task, it must be removed from the scheduling pool and its context must be established. Again, the serial bottlenecks associated with a common scheduling pool introduce overheads. Having a separate queue for each task group helps reduce this overhead. Pre-scheduling tasks can completely eliminate serialization but creates load imbalance. These issues will be discussed in more detail in section IV-A.

When a task completes execution, its context has to be disestablished and control has to be returned to the scheduler. Moreover, task completion needs to be signaled for synchronization purposes and this introduces yet another (potential) source of serialization. Simple and efficient techniques are required to minimize synchronization overheads. Barrier synchronization is generally used to synchronize threads of control at a common point. It is possible to design "lock-free" barriers, however it does not appear possible to design a constant-depth barrier without the use of specialized hardware.

All of the above mentioned sources of overhead can be detrimental to the performance of a parallel program. They can limit flexibility by increasing the granularity required for efficient parallelism or by imposing an excessively static style of programming. Efficient mechanisms must be developed to exploit medium-to-small grained parallelism and allow program parallelism to be expressed naturally. Such mechanisms can not only be used by programmers to indicate program parallelism explicitly, but also by a parallelizing compiler. In the following section we will review some of the task management schemes proposed for parallel loops and their relationship to general task scheduling problems.

III. PARALLEL LOOP SCHEDULING VS. HETEROGENEOUS PARALLELISM

Since loops have regular structure, they are relatively easy to schedule dynamically. The simplest dynamic loop scheduling algorithm is called self-scheduling and consists of each processor fetching an iteration of the loop one at a time by (atomically) incrementing the shared loop index. Self-
scheduling performs the best possible load balancing but incurs the greatest overhead due to serialization of access to the loop control variable for each loop iteration.

A variant of self-scheduling, called chunk-scheduling or block-scheduling, reduces overhead by having processors schedule blocks of iteration at once. The scheduling and synchronization overhead is reduced by the block size but load balancing is compromised. If the chunk size is $k = \lceil N/P \rceil$, where $N$ is the number of iterations and $P$ is the number of processors, then block scheduling is the same as pre-scheduling or static-scheduling.

Guided self-scheduling (GSS) [13] combines the best of the previous two approaches. It dynamically changes the number of loop iterations assigned to idle processors. Scheduling large chunks of iterations in the beginning results in low scheduling overhead, while scheduling small chunks towards the end provides good load balancing.

Beckmann and Polychronopoulos [4] have studied the effects of self-scheduling, GSS, and pre-scheduling algorithms on the speedup of parallel loops. Their asymptotic results show that pre-scheduling is the best strategy until variation in loop execution times becomes large, after which self-scheduling yields best speedups. These results can be intuitively understood in terms of adjustment of task granularity (intervals between synchronization points). It has been shown in [18] that multiprocessor performance strongly depends on the ratio $R/C$, where $R$ is the length of a task and $C$ is the length of overhead produced by that task. The ratio expresses how much computation is performed per unit of overhead. Speedup of a parallel program is directly proportional to this ratio. For small variations in task times, the value of $C$ is relatively fixed. Parallel loops provide ample opportunity to adjust the value of $R$. Pre-scheduling gives the highest value of $R$ (and hence a high $R/C$ ratio) and self-scheduling gives the lowest value of $R$ (and hence the lowest value of $R/C$), whereas GSS gives an intermediate value. For large variations in task times, the value of $C$ for pre-scheduling increases more than it does for GSS or self-scheduling due to poor load balancing produced by pre-scheduling. The speedup curves derived in [4] follow directly from this simple argument.

Whereas parallel loops provide freedom in adjusting task granularities by grouping multiple iterations into a single task, the generally more constrained data dependencies between heterogeneous parallel subcomputations of a program limit the ability to arbitrarily adjust task granularities. Performance of such programs is not strictly governed by issues of static versus dynamic scheduling, and the results of [4] cannot be directly applied. Instead, alternative approaches must be carefully compared with respect to the overheads identified in section II. Comparisons between approaches must consider their relative ability to efficiently manage tasks at the same level of granularity.

In the following section, we present the results of an empirical study involving a real application. The application exhibits a large amount of subroutine-level parallelism. However, the synchronization requirements severely constrain the ability to adjust task granularities. This study allows us to compare various task management strategies, needed to exploit such form of parallelism.

IV. AN EMPIRICAL STUDY

The study was based on variational recursive dynamics simulation of a typical four wheel vehicle [3,10]. Recursive dynamics formulations require solution of highly non-linear dynamical algebraic equations. They involve large amounts of scalar floating point computations and operations on small dimension structural data, most of them on vectors or matrices of dimension no more than six. They offer little or no opportunity for vectorization or loop-level parallelism. (In fact, the parallelizing and vectorizing compiler on the Alliant FX/8 computer system could extract a maximum speedup of only 1.10.)

Although highly scalar, recursive dynamics formulations are well-suited for task-oriented parallel processing, at the subroutine-level [10]. Several observations are to be made concerning the application:

(i) The computational threads converge and synchronize between various phases of analysis, and hence parallelism is naturally expressed as synchronous parallel subroutines.

(ii) The number of concurrent tasks range between five and fifteen and the granularity of these tasks range from tens of instructions to few hundreds of instructions. This means that there is little opportunity to adjust task granularities. Since task granularities are relatively small, efficient low-overhead mechanisms are needed to effectively exploit the available parallelism.

(iii) By measuring individual task execution times, the maximum achievable speedup for the application can be analytically determined. Such a speedup curve is plotted in Fig. 1. The effectiveness of a task management mechanism can be gauged in terms of how close it comes to achieving this maximum.

Since recursive dynamics formulations are tightly-coupled and share large amounts of data, the best-suited computing environment is a shared-memory multiprocessor system. Experiments were carried out on an eight processor Alliant FX/8 minisuper computer system and a BBN Butterfly. The study compared three parallel processing support environments: (i) a general-purpose queue-based tasking environment, such as Schedule [7], (ii) operating system assisted task scheduling provided by the Mach operating system [5], and (iii) a specially developed environment for efficient management of synchronous groups of subroutines.

In addition, since the Alliant FX/8 has special purpose hardware to support self-scheduling of loop iterations [1], it was possible to compare the efficiency of software-based support mechanisms with that of specialized hardware support.

In designing a specialized parallel processing support environment to implement the recursive dynamics code, various scheduling and synchronization schemes were considered. One obvious possibility was to design a queue-

\[ \text{Butterfly is a registered trademark of BBN Advanced Computers, Inc.} \]
based scheduling mechanism, similar to ones used in several general-purpose multitasking packages. Initial measurements of task granularities and the overheads due to bottlenecks associated with serial-access queues clearly indicated that any \( O(n) \) task scheduling and startup mechanism would severely limit our ability to extract the fine-grain parallelism of the application.

An alternative to dynamic queue-based scheduling is the static placement of tasks on computational threads. This would reduce the scheduling overhead to zero (although barrier synchronization overheads would remain). However, as discussed in section II, this approach was impractically difficult with respect to coding and maintaining the application, and would not allow run-time selection of the number of processing elements to be used. Design issues and techniques related to a tasking environment that efficiently supports hierarchically nested synchronous subcomputations are described in the following section.

IV-A. AN ULTRA LIGHT-WEIGHT SYNCHRONOUS TASKING ENVIRONMENT

In section II, we have identified four major sources of overheads in a general tasking environment. In this section, we will discuss techniques and methods used in designing a specialized synchronous tasking environment that minimizes or eliminates these overheads.

In our tasking model, a task can be scheduled for execution by another (parent) task and it can schedule other (child) task(s) for execution. In fact, a set of synchronous tasks, called a task_group, can be scheduled by a single operation.

To reduce task creation overhead, tasks and task_groups are defined once (prior to their first invocation) and can be invoked any number of times during program execution. Tasks are written as ordinary subroutines and may have any number of parameters. Since a task is defined only once, actual arguments for its parameters must be given at task initialization time.

Since we require that tasks and task_groups be initialized before their first invocation, we can use this information to pre-schedule tasks onto an associated task_queue. When a task_group is ready for execution, its task_queue can simply be "dispatched" by the parent task. Thus a task_group can be launched via a single, constant-time call. Moreover, such a mechanism can naturally implement the synchronous behavior of our programming model. That is, the call returns only when all of the child tasks have finished their execution.

Each group of synchronous tasks (i.e. a task_group) is assigned to a processor set for execution. Processor sets are disjoint partitions of the available number of physical processors and must be defined before any task_group can be initialized. Many task_groups can be assigned the same processor set. However, to avoid contention, we require that no more than one task_group be dispatched for simultaneous execution on the same processor set.

When a task_group is scheduled for execution, the processors in the processor set access its task_queue to obtain a task. An initial task assignment can be done at task_group initialization time. Each processor executes its initial task and then checks to see if there are more tasks to execute. Additional tasks are removed from the task_queue under mutual exclusion. The mutual exclusive access to this queue is the only serialization part in the entire tasking environment. This dynamic, self-scheduling approach achieves better load balancing but introduces some serial bottlenecks.

Alternatively, if task execution times can be estimated prior to task_group initialization, we can statically distribute the computational load among the processors and thus eliminate the serialization associated with dynamic self-scheduling. Static assignment schemes, such as bin-packing or a simple "round-robin" heuristic, can be used to assign tasks to processors. Processors can obtain a task to execute without mutual exclusion but load balancing becomes sub-optimal.

Finding optimal schedule is a difficult problem which becomes even more difficult when hierarchical structures are involved. The nested nature of the nested task makes its scheduling a much harder problem to deal with and has received little attention in the classical scheduling theory [6].

No good heuristic exists for the general problem. To compare the relative efficiency of static versus dynamic scheduling, the environment was designed so that the user can specify which type of scheduling should be employed when initializing a task_group.

We use a barrier synchronization mechanism similar to the one described in [2]. The entry phase of the barrier is linear in the number of processors in a processor set. The exit phase uses a broadcast mechanism that allows us to implement constant-time dispatching of a task_group onto its processor set.

IV-B EXPERIMENTAL RESULTS

The vehicle dynamics simulation algorithm was implemented using the specialized tasking environment described in section IV-A. Eight task_groups were identified, with a total of 73 tasks. The granularity of these tasks, as measured on an Alliant FX/8 CE (computational element), ranged from 90 microseconds to 2.22 milliseconds. One minute of vehicle motion was simulated constituting a total of 10260 iterations of the algorithm. A high resolution clock (10 microseconds per tick) was used to measure the elapsed computation time. Timing calls were placed before and after the main simulation loop. Average time per iteration was then obtained by dividing the total elapsed time by total number of iterations. The serial execution time was measured without any tasking calls.

The simulation was implemented using both the static and dynamic scheduling options of the ultra-light weight synchronous environment. The objective was to study the effect of load imbalance due to static scheduling and the effect of serialization due to dynamic scheduling. The resulting speedup curves are presented in Fig. 1. The performance gap between the two scheduling schemes is not very significant. However, the dynamic scheduling performs better in this case. This is due to the sub-optimal schedule generated by the static scheduling heuristic. The heuristic works well if the execution times of tasks do not vary significantly. Some hand-generated test cases showed that static scheduling can

211
sometimes performs better than dynamic scheduling under such circumstances.

The important observation is that both schemes successfully extracted in excess of 80% of the available parallelism (88% by dynamic scheduling and 83% by static scheduling) for an application with not more than 15 concurrent tasks, each with an execution time of about few hundred microseconds. The efficiency is a direct consequence of utilizing the right support mechanisms that match naturally with the programming model.

For comparison purposes, the simulation algorithm was parallelized using a general queue-based tasking environment, namely the Schedule package [7]. As can be seen from Fig. 2, the overheads associated with general tasking operations limit the speedup of the vehicle dynamics algorithm to about 2 for an eight processor system, i.e. only 36% of the maximum achievable speedup. Some of these overheads can be attributed to the run-time creation of tasks (which involves creating context for the task and placing parameters on the task descriptor), serialization due to a global task queue, and providing various forms of synchronization and inter-task communication.

Recently, some operating systems for parallel processor computers have begun to provide mechanisms to support various forms of parallelism, including heterogeneous threads. In particular, the micro-kernel of Mach operating system provides various light-weight synchronous and asynchronous thread management functions. To gauge the effectiveness of these mechanisms, the application was implemented on a BBN TC2000 which uses a derivative of Mach's light-weight task management routines [16]. The speedup curve is plotted in Fig. 2. The effect of using better-suited mechanisms on the program speedup is clear. Still, overheads due to operating system involvement limit the performance to about 50% of the maximum possible speedup.

Since the Alliant FX/8 has sophisticated hardware mechanisms for support of loop-level parallelism, it was possible to compare the performance of hardware versus software management mechanisms for this application. The dynamics code was parallelized using the specialized hardware-based mechanisms of Alliant FX/8 computer system. Because the parallel subroutine invocations in the example application are not nested, synchronous group of subroutines can be launched in a self-scheduling fashion by calling subroutines out of successive iterations of a do-loop. This allows the Alliant hardware for loop parallelization to be used to implement the dynamics code. The resulting performance is plotted in Fig. 3. It is interesting to note that the purely software-based scheme is able to perform as well as or better than the hardware-based implementation for the studied application.
V. Conclusions

The results of the empirical study indicate that the choice of an appropriate run-time parallel processing support mechanism can have a dramatic impact upon the ability to successfully extract heterogeneous parallelism from programs. In particular, the efficiency (lack of overhead) of the support mechanisms become critically important as the granularity of subcomputations become relatively fine.

The efficiency of a support mechanism goes well beyond the choice of scheduling policy. Indeed, similar performance was obtained in our study for mechanisms employing static vs. dynamic task scheduling. Implementing efficiency must, in general, be obtained through restricting the functionality and generality of the mechanisms to the minimal amount necessary for the specific application. The generality of a support mechanism, in terms of structuring flexibility, range of synchronization and communication options, degree of preemptive control, etc. tends to dictate its basic overheads even in applications where these features are not utilized.

Our results indicate that reasonably general-purpose support mechanisms are not likely to be efficient enough to support relatively fine-grained heterogeneous parallelism. This suggests that parallelizing compilers should provide a range of appropriate run-time support mechanisms rather than relying on one set of general operating system, application, or hardware-level mechanisms.

It is also apparent that syntactically closed constructs such as those described in the previous section have the greatest potential for extremely efficient implementation. Thus we believe that parallelizing compilers should attempt to formulate the parallel structure of the application in a manner amenable to the use of these constructs. The exact nature of comprehensive set of run-time support mechanisms is not apparent at this time. Many complex issues related to task graph restructuring, multilevel scheduling of heterogeneous subcomputations, etc. must be addressed before this question can be completely answered. These issues are important enough to warrant serious attention.

Acknowledgements

Experiments reported in this work were carried out on computing facilities at Center for Computer Aided Design, University of Iowa, and ACRF, Argonne National Labs.

References