LA-UR- 98-1350

Approved for public release; distribution is unlimited.

Title: Dependence Driven Execution for Multiprogrammed Multiprocessor

CONF-980767--

Author(s): Suvas Vajracharya
DirkGrunwald

Submitted to: International Conference in Supercomputing 1998

Los Alamos
NATIONAL LABORATORY

DISCLAIMER

Dependence Driven Execution for Multiprogrammed Multiprocessor

Suvas Vajracharya
CIC/ACL
Los Alamos National Laboratory
Los Alamos, NM, U.S.A.
suvas@lanl.gov

Dirk Grunwald
Department of Computer Science
University of Colorado
Boulder, CO, U.S.A.
grunwald@cs.colorado.edu

Abstract *Barrier synchronizations can be very expensive on multiprogramming environment because no process can go past a barrier until all the processes have arrived. If a process participating at a barrier is swapped out by the operating system, the rest of participating processes end up waiting for the swapped-out process. This paper presents a compile-time/run-time system that uses a dependence-driven execution to overlap the execution of computations separated by barriers so that the processes do not spend most of the time idling at the synchronization point.*

Keywords: Run-time systems, multiprogramming, loop scheduling, dependence-driven execution, barrier synchronization, coarse-grain dataflow.

The parallel execution of a sequence of loop nests is typically broken into phases, each phase consisting of a simple loop separated by a barrier synchronization to ensure that the data dependencies between phases or a collective operation are respected. A barrier synchronization insists that all the participating processes be collectively done with a particular phase of a computation before the next phase is begun. This is undesirable in that the barrier synchronization causes the computation to stall while waiting for the *slowest* process. On multiprogrammed multiprocessors, a participating process may be swapped out to run a process from a different job, making the problem worse. Figure 1 shows that the swapped out processes effectively lengthen the time required to achieve a barrier.

Much of the work on multiprogrammed environment has been done on the operating system or combination of user space/operating system. Several researchers [1, 2, 10, 11] have focused on ways to establish some cooperation between the application and the operating system kernel. By extending the kernel to communicate with the application, the kernel can avoid pre-empting a task that is in a critical section. However, because these methods require modification to the operating system and more importantly, because they require that the applications handshake with the O.S. scheduler, many commercial operating systems do not support scheduler activations and scheduler-conscious synchronization.

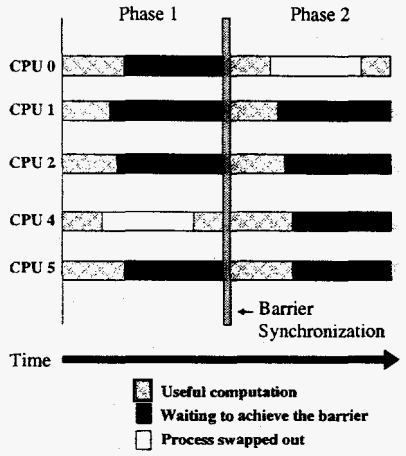


Figure 1: Performance Degradation in Multiprogrammed Multiprocessors

Others have focused on various multiprogramming policies for scheduling kernel processes such as unsynchronized time-sharing (time-slicing), synchronized time-sharing (co-scheduling) [14], and space-sharing (hardware partitions) [16]. It is extremely difficult to find a single policy that can maximize utilization, ensure fairness, and, at the same time, keep the overheads low.

We take an approach that is not incompatible with the above methods. We are presenting a user-level loop scheduling method that adapts to dynamic increases or decreases in the available number of CPUs without requiring modifications to the operating system. This makes the runtime system that we are proposing portable and easy to use, and makes no assumptions about the policies or the kinds of extra services that the operating system supports.

The loop scheduling method proposed here extends traditional loop scheduling methods such as static, affinity and dynamic loop schedulers by using symbolic data dependence information. Having dependence information allows the run-time system to schedule the entire loop structure consisting of several simple loops. To illustrate, let us turn to figure 1 once more. Using

a dependence-driven execution, some of the processors can begin computing phase 2 while the other processors are still computing phase 1. This allows us to do useful computation while waiting for processes that the operating system has swapped out. However, overlapping executions of different phases is only legal if the computation respects the data dependencies between the two phases. In our companion papers [18, 17], we discussed how DUDE system improves memory locality and increases parallelism. In this paper, we examine how DUDE performs in a multiprogrammed environment.

1 The DUDE Runtime System

In this section, we describe DUDE runtime system. DUDE is meant to be used either as a target for optimizing compilers, or as a set of library calls that programmers use directly to optimize their code. The intent of DUDE is to optimize the common case in scientific and engineering applications which consists of loop nests, each loop nest ordinarily implementing a collective operation on a multi-dimensional array.

To enforce dependencies between consecutive loop nests (inter-loop dependencies), conventional models of data parallel computation insert a barrier synchronization between them, dividing the computation into distinct phases such that each loop nest is completed in its entirety before the next loop nest is begun. This model of computation introduces false dependencies or dependencies that were not in the semantics of the original program. By using a dependence-driven execution, DUDE overlaps the execution of iterations from different loop nests to increase parallelism and improve memory locality. The increased parallelism comes from the asynchrony of the model: Processors need not idle, while waiting for other processors stuck at a barrier to complete a particular phase of the computation; they can proceed to the next phase if the dependencies on those iterations are satisfied. The improved locality that the model offers, comes from applying multiple operations (from different loop nests) on an array region before that array region leaves the processor's cache.

An optimizing system can also use a dependence-driven execution to increase the parallelism of a single loop nest with dependencies within the iterations of the loop (intra-loop dependencies) or a doacross loop. In DUDE, the system schedules the iterations within a loop nest the moment the dependence constraints on them are satisfied, similar to data-driven models.

The basic model has some similarities to the underlying concept of systolic arrays. Like systolic arrays, computation in DUDE consists of a large number of processing elements (cells) which are of the same type. However, for efficient computation on general-purpose computers, the granularity of computation in DUDE is much coarser. In our implementation, these cells are actually C++ objects consisting of an operation and a descriptor describing a region of data to which the operation is applied. Thus, the granularity of the processing elements is blocks of iterations. We term these objects *Iterates*, and an array of these Iterates makes up an *IterateCollection*.

To describe the runtime system, we first describe how the user describes the static loop structure. We then explain how the runtime system takes this loop description to effect a dependence-driven execution.

1.1 Static Description of Loops in DUDE

The semantics of loops in imperative languages overspecify the order in which iterations are to be executed. Since a desirable order may not be known until runtime, we want to describe loops with the weakest restrictions on the order while preserving the semantics of the original loop, such that we do not commit to any specific iteration order at compile-time. Although the loop structures that appear in a real program can be quite complex (conditionals, varied levels of nesting, etc.), the constructs that are used to build and define the loop structures themselves are quite simple. For this reason, we have taken a modular approach to describing complex loop structures by providing simple objects as building blocks. In DUDE, data, blocks of iterations, dependence rules, loop bodies, and loop nest structures are objects which can be put together to describe complex loops. By putting together and specializing these objects, the user specializes the system to create a "software systolic array" for the application at hand. This object-oriented model is based on AWESIME [6] and the Chores [2] runtime systems. The following is a list of objects in DUDE:

- **Data Descriptor:** Data Descriptors describe a sub-region of the data space. For example, the system can divide a matrix into sub-matrices with each sub-matrix being defined by a data descriptor. On a two dimensional sub-matrix, the methods on this object, SX(), EX(), SY(), and EY() retrieve the corners of a two-dimensional sub-matrix.

- **Dependence Rule:** Dependence rules summarize all dependencies between iterations in a loop structure. The rule fires when a completed Iterate's ID, I matches the left-hand side of the rule to produce a list of Iterates that the system may now enable as a result of I having completed.

- **Iterate:** An Iterate is a tuple <data descriptor, operator> representing a non-blocking continuation which runs on the stack of the worker threads. An Iterate defines the granularity of the parallel execution of a loop. The user specializes an Iterate by overloading the default operator with an application-specific one consisting of statements found in the body of a simple loop. The system applies the virtual operator to the data described by the descriptor. Each Iterate has a unique ID which the system instantiates with the left-hand side of a dependence rule to produces candidate Iterates (right-hand side of the dependence rule).

- **IterateCollection:** An IterateCollection, as the name implies, is an array of Iterates. An IterateCollection represents a single loop nest (or a collective operation) that performs a simple operation on the entire data space. The dimensionality of an IterateCollection is normally the same as that of the data array on which it operates. Figure 2 shows the relationship between Iterates and IterateCollection. Defined as a template, users can

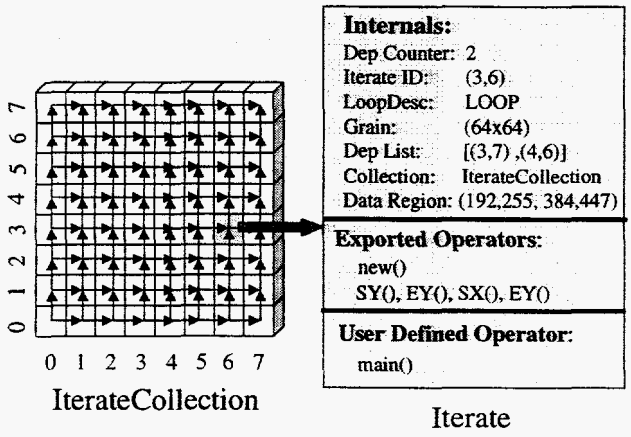


Figure 2: Building Blocks of Loops: Iterates and IterateCollections

choose different types of IterateCollection templates to specify the scheduling behavior that is to be applied to the loop nest. The following is a list of some of the C++ templates defined in DUDE: *QuadtreeCollection*, *MortonSeqCollection*, *AffinityCollection*,and *StaticCollection*.

- **LOOP:** A LOOP is a template structure used to describe a sequence of collective operations by putting together one or more IterateCollections. The compiler uses the following methods provided by the LOOP object to put together different IterateCollections:
  - *SetOp(n,IterateCollection)*makes the IterateCollection the nth collective operation in a sequence of collective operations.
  - *SetDependence(IC1, IC2, dep)* defines the symbolic dependence *dep*, from IterateCollection $IC1$ to $IC2$. The compiler dervies the symbolic dependencies from array subscript expressions in assignment statement of the two loops. Note that if $IC1$ is the same as $IC2$, then this describes a doacross loop.
  - *Execute()* executes the entire loop nest described by the loop descriptor.

As shown in figure 2, an Iterate has pointers to both the IterateCollection to which it belongs, and to the loop descriptor to which the IterateCollection belongs. This allows a particular Iterate to derive information about the entire loop structure. Endowed with this information, an Iterate can determine what is the continuation (what is the Iterate in the subsequent IterateCollection), what are the loop bounds (to determine whether the entire loop should be terminated), and what is the scheduling behavior that is to be used in choosing the next Iterate to run. Therefore, each Iterate has sufficient information to act locally to schedule the entire loop structure globally.

1.2 Generating Symbolic Dependence Rules

We have described how the system uses dependence rules in our runtime system, but we have not described how to the user derives them from the source loop. We will restrict our discussion to the loop structure for a one-dimensional Red/Black SOR, although a compiler/runtime system can apply a dependence-driven execution to more complicated loop structures. We will then give an example of how to derive a dependence rule for a specific instance of this loop structure.

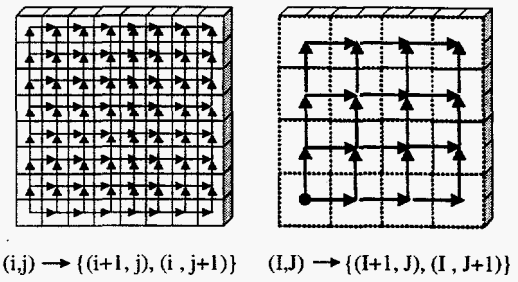


Figure 3: Micro and Macro Dependence Rules

1.3 Macro and Micro Dependencies

We will call dependencies between two iterations, or two points in an iteration space, *micro-dependencies*, and dependence between Iterates, *macro-dependencies*. What is the relationship between macro and micro dependencies? This is an interesting question because our implementation requires macro-dependence rules, whereas true dependencies are in terms of micro-dependencies.

The system can easily derive macro-dependence rules from micro-dependencies if the array index sub-expressions of statements in the loop body are linear or affine functions of the induction variables. In fact, since the mapping between iteration space and Iterate space is itself linear, the relation between the source and target of the dependencies in the Iterate space is isomorphic to the relation between dependencies between points in the iteration space, as shown in figure 3. Fortunately, affine subscript expressions are also the most common form of subscripts in loops [19]. For linear subscript expressions, we can derive the macro-dependence rule directly from the sub-expressions: we simply replace the iteration index i which ranges over $1..N$, with an Iterate index which ranges over $1..N/g$ where g is the granularity of Iterates.

To illustrate, we now look at the process of deriving macro-dependence rules with respect to a one-dimensional Red/Black SOR, although this method applies to applications with more complex dependencies. The loops in the Red/Black SOR has following form:

```
DO t = 1, T
     DOALL i = 1,N-1 by 2
     A[i] = (A[i − 1] + A[i + 1])/2
     ENDDO
     DOALL i = 2, N by 2
     A[i] = (A[i − 1] + A[i + 1])/2
     ENDDO
ENDDO
```

The micro-dependence rules are:

$$i^d \rightarrow \{i^u - 1, i^u + 1\}$$

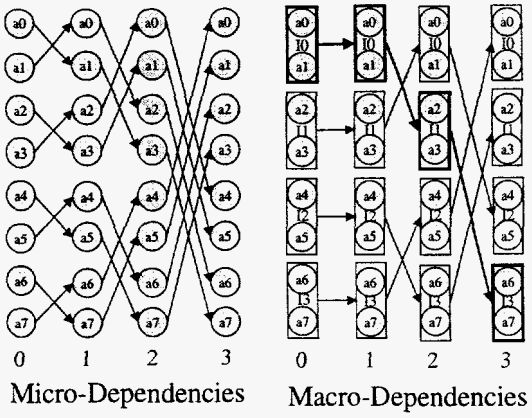


Figure 4: The Butterfly Circuit

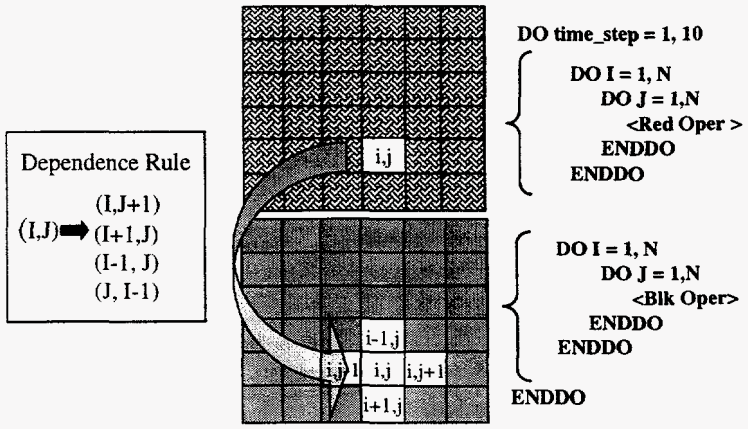


Figure 5: Multi-Loop Dependencies on Red/Black SOR

$$i^u - 1 \rightarrow i^d \; or \; i \rightarrow i^d + 1$$
$$i^u + 1 \rightarrow i^u \; or \; i \rightarrow i^u - 1 i^d \rightarrow i^d$$

Converting to macro-dependence rules and simplifying by removing redundant terms, we have the macro dependence rule:

$$I^d \rightarrow \{I^u - 1, I^u, I^u + 1\}$$

1.4 Non-linear subscript expressions

Although, linear subscript expressions cover the majority of loops found in real applications, there are few important applications that have non-linear expressions. For some important class of loops structures, we can create a library of circuits to implement a dependence driven execution. For example, a common dependence pattern between consecutive collective operations of a large class of applications is the butterfly pattern shown in figure 4. This figure also shows that the relationship between micro-dependencies and macro-dependencies is not one-to-one. Nevertheless, a micro-dependence rule can be "hand" generated for this application. The bold squares on the right side of the figure indicate a possible path of a dependence-driven execution for Iterate $I0$.

1.5 Example

To see how all this fits together, we show how a user may describe the Red/Black SOR application, which consists of two collective operations—the *Red* operation and *Blk* operation. Figure 5 shows the original code and the inter-loop dependencies. Both the *Red* and the *Blk* operations in the loop body simply take the average of an element's neighboring point creating the dependence shown in the figure. Note that since the loops for the *Red* and *Blk* collective operations are not nested within each other, they are not perfectly nested, and hence, unimodular transformation does not apply. Furthermore, because of the backward dependencies in this application, loop fusion does not apply either. Figure 6 shows the application as it would appear when written for DUDE. For this application, there are two Iterates, the *RED* and the *BLK*, with corresponding *BLK::main* and *RED::main* methods that overload the operator to specialize the Iterate for this application. These Iterates compose the *RedColl* and *BlkColl* collections. Finally, these collections themselves are combined in the loop descriptor to create a sequence of collective operation that iterates up to 10 iterations.

1.6 Execution of Loops on the DUDE System

Having described how a user specifies a loop structure to the runtime system, we now turn to how the runtime system takes the static specification of a loop structure and execute it using a dependence-driven model. Figure 7 shows the basic model that the runtime system uses. Initially, the system pushes only the unconstrained Iterates onto the system work queue. This allows idle processors to remove an Iterate from the work queue to perform the operation associated with that Iterate. The completion of the operation terminates the activation of that Iterate, which may satisfy dependence constraints to other Iterates based on what the data dependencies are and what other Iterates have completed. The dependence satisfaction engine creates works while the scheduler distributes the work for different processors to complete. This creates a cycle shown in figure 7, which the system repeats until the entire loop nest is completed.

Figure 2 shows the structure of the Iterate object. Internally, each Iterate contains a counter representing the number of dependence arcs which sink into that Iterate. When an Iterate completes, the system matches the ID of that Iterate to the left-hand side of the dependence rule, which when fired, produces a list of sinks of dependence arcs; that is, it produces a list of Iterate ID's which can be used as indices to the IterateCollection. The system then decrements the counter associated with the sink Iterate. If the counter becomes zero, then the system can enable that sink Iterate for execution.

Dependence-driven execution introduces runtime overheads that do not directly contribute to the numerical computation. To make the implementation as efficient as possible, the system initializes the internal dependence representation of the entire loop structure, such as the dependence counters, prior to the execution of any iterations. Thus, the cost of the initialization which is done only once, is amortized over several iterations of the loop. If the list of sinks emanating from an Iterate

```
void main() {
  RedColl = new IterateCollection<RED>(A,dim,BLOCK,cpus,grain);
  BlkColl  = new IterateCollection<BLK>(A,dim,BLOCK,cpus,grain);
  LOOP loop(iterations = 10);
  loop.SetOp(0,RedColl);
  loop.SetOp(1,BlkColl);
  loop.SetDependence(RedColl,BlkColl,
                     "(i,j) => (i+1,j),  (i-1,j),  (i,j+1), (i,j-1)");
  loop.Execute();
}
```

```
void RED::main() {
  for (I=SX(); I <EX(); I=I+2)
    for (J=SY(); I<EY(); J=J+2)
        A(I,J)=AvgOfNeigbhors(I,J)
}
```

```
void BLK::main() {
  for (I=SX()+1; I <EX(); I=I+2)
    for (J=SY()+1; I<EY(); J=J+2)
        A(I,J)=AvgOfNeigbhors(I,J)
}
```

Figure 6: Red/Black SOR on DUDE

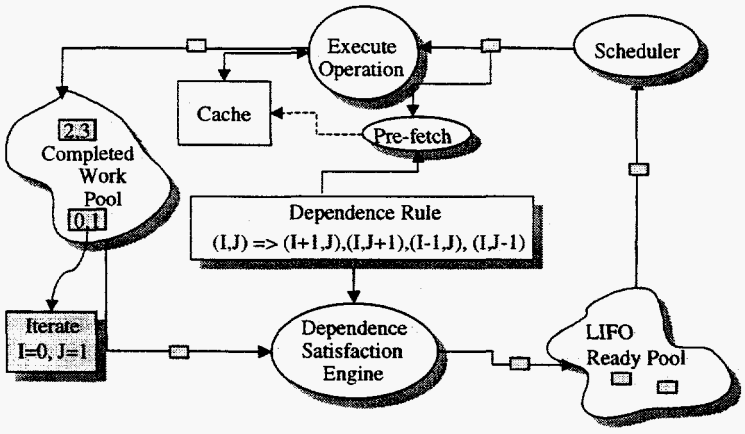
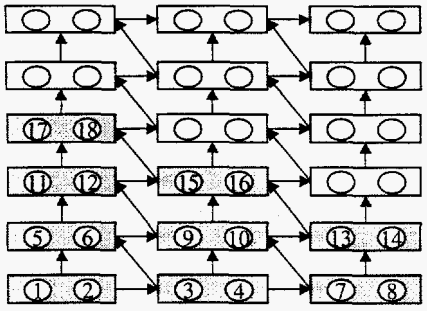


Figure 7: Dependence-driven Execution Model



Dependence Rule: I ➡ (I-1), (I), (I+1)

```
PDE::main {
    for (I = SX(); I <= EX(); I++)
        A[I+1] = 1/3 * (A[I] + A[I+1] + A[I+2]);
}
```

Figure 8: Iteration order for Hyperbolic 1D pde using DUDE

do not change during the execution of the loop, as in static and affinity scheduling, the system can create the list for each Iterate in the IterateCollection during loop initialization.

We now continue with the example of Red/Black SOR to describe how the system executes the loop specification. The *Execute()* function of *loop* in figure 6 starts the system by pushing all of the initially unconstrained Iterates onto the system work queue. The initially unconstrained Iterates in this example consist of Iterates from the *RedColl* collection. Now the computation begins. After the initial Iterates have been loaded, a worker pops off a (*Red*) Iterate from the system work queue and applies the *main* operator to the data described by the descriptor for that Iterate. On a parallel execution, workers may steal work from a different processor to balance the workload. When completed, the system determines the list of sinks of the dependence arcs for that Iterate based on the dependence rule. For each sink, the system decrements the counter in the destination Iterates which, at this point in the execution, are *Blk* Iterates. If the count is zero, the Iterate becomes unconstrained, or enabled, and the dependence satisfaction engine pushes these enabled Iterates onto the system work queue.

Continuing, the completion of a *Blk* Iterate can further enable a *Red* Iterate from second time step, and so forth. This describes a depth-first traversal of the iteration space, since a *Blk* operation can begin before all of the *Red* operations are completed.

In DUDE, Iterates represent a sub-computation on a sub-region of data. The fact that the granularity of the processing elements, or Iterates, does not consist of a single iteration but blocks of iterations, raises some questions. What does each Iterate compute? In which order should the system computer the iterations within an Iterate; that is, what is the intra-Iterate order? In which order should the system compute the Iterates themselves; that is, what is the inter-Iterate order? The intra-Iterate order is simply the order of the traversal used in the original loop except that the loop bounds are limited to the data region to which the Iterate has been delegated. The inter-Iterate order, however, is determined by the dependence-driven execution.

DUDE system consists of three types of dependence-driven loop scheduling methods: static, affinity, and dynamic. A description of these schedulers in DUDE can also be found in our paper [17]. Depending on the application, or based on runtime conditions, the user can choose one of these methods by instantiating the appropriate IterateCollection template which determines the scheduling behavior. In static scheduling, the scheduler decomposes data into fixed-sized chunks, and distributes the data to different processors prior to the execution of loop iterations. In affinity scheduling, the system decomposes the data prior to the loop execution but the scheduler distributes the decomposed data during the execution of the loop. Finally, in dynamic scheduling, the scheduler decomposes and distributes the data during the execution of the loop. We begin by discussing static and affinity scheduling.

1.7 Static and Affinity Scheduling

In both static (instantiation of the StaticCollection template) and affinity scheduling (the AffinityCollection template), the system associates an Iterate with a home CPU. In static scheduling, an Iterate is permanently bound to the home CPU for the duration of the program, while in affinity scheduling, the home CPU is the preferred place where the Iterate will be executed but the Iterate may be stolen by a different idle CPU to balance the load. The advantage of using static scheduling is that having a fixed home for Iterates allows the compiler/runtime system to determine which Iterates need to communicate with a remote CPU and which Iterates need only local communication within a CPU. Iterates that do not need to communicate with another CPU also do not need to use locks for accessing local work queues. The advantage of affinity scheduling is that the scheduler can adapt to the dynamic fluctuation of the workload by distributing Iterates to idle processors while trying to maintain data locality [12].

One of the parameters used in instantiating a StaticCollection or an AffinityCollection is the decomposition method, which determines how the system divides the data between the different Iterates. The user chooses the decomposition method such as by BLOCK or CYCLIC similar to decomposition and distribution utilities available in HPF [9] and pC++ [3]. One important difference, however, is that in the process of data decomposition, DUDE takes flat data and creates objects or Iterates, which are tuples consisting of both data and operation.

1.8 Dynamic Scheduling

In dynamic scheduling, the data decomposition and data distribution is determined at run-time by the scheduler. The definition of loop structure and dependence specification, however, remains similar to static and affinity scheduling. Unlike static and affinity scheduling, the chunk sizes are not fixed, but vary during runtime as the scheduler sees fit to balance the workload.

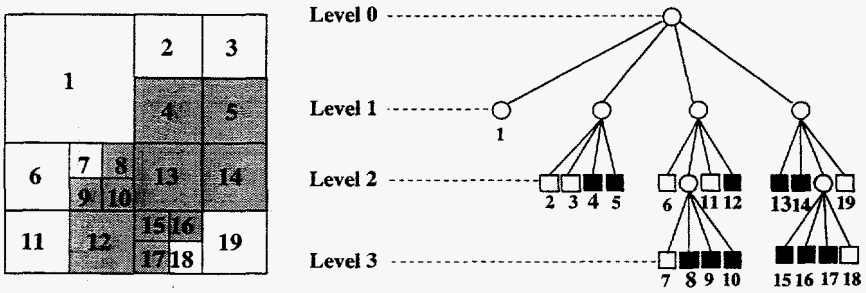


Figure 9: A Quadtree

In DUDE, we use spatial data-structures based on the principle of recursive decomposition called *quadtrees* (or its three dimensional version, *octree*) breakdown and coalesce neighboring Iterates. Figure 9 shows an example of a *quadtree*. The tree in the figure represents the Iterate space on the left side of the figure. The dark portion of the Iterate space (which corresponds to dark nodes in the tree) represents the Iterate space region that is ready to be executed or enabled. The resolution of the decomposition(*i.e.*, the depth of the tree) can be fixed or varied during run-time. Note that the quadtree-representation numbers the nodes such that the children of any node compose a contiguous block.

In the *quadtree* decomposition, each subdivision is a block divided into four equal parts. Alternatively, at each subdivision, the block could be divided into two parts as in a *bintree*. Samet [15] gives a comprehensive description of these data-structures. These structures enables efficient indexing into the pools of Iterates because the chunk sizes are uniform at any given level. Given the level and the index, the system can retrieve the Iterates from a pool by simply using random access. Similarly, unification with the left-hand side of the firing rules is efficient since, at any given level, the unification algorithm is the same as it would be for static decomposition where grain sizes are uniform.

The hierarchical structure is also ideal for dealing with the problems of coalescing small chunks. As the dependence-driven execution enables a group of neighboring small chunks of iterations, the system can coalesce them to create a larger set of contiguous iterations. This amounts to setting the parent node when the system has enabled all of its four child nodes at the finer resolution.

We now describe the quadtree scheduling method can which consists of the following steps:

1. Construct a quadtree for the iteration space using some minimum granularity; that is, determine the granularity of the leaf nodes.

2. During runtime, dynamically determine the scheduling granularity depending on the amount of iterations left to do.

3. Assign iterations to processors at this granularity.

4. When iterations are completed, determine which new iterations can be enabled. If new iterations can be enabled, see if the total set of enabled iterations can be coalesced into large blocks. Coalescing amounts to marking the parent node as enabled if all its children are enabled.

5. Go to step two until no new iterations can be enabled.

2 Related Work

Using dependence information to increase parallelism is not a new idea. One issue that distinguises the various works is the unit or the granularity of parallelism. A very fine granulaity of asynchronuous computation requires a significanlty different design than for a coarse-grain computation. For example, dataflow computers such as the Manchester Dataflow Machine [7, 8] relied on special hardware to orchestrate parallelism at the level arithmetic operations. In Mentat [5, 4], the unit of parallelism were functions. In the autoscheduling work by Moreira and Polychrnopolus [13], the unit was a task which may consist of a entire loop. These loops might be scheduled by guided-self scheduling across the available processors so that the iterations of a doall loop can be run in parallel. Iterations from different consecutive loops are not executed in parallel, but computed in lock step by using a barrier. Put differently, the nodes in the dataflow graph is at the granularity of entire loops. The granularity of parallelism in the DUDE runtime system is somewhere between the fine grain computation in dataflow machines and coarse grain macro-dataflow approach in. Another dimension along which to compare the different work is the extent to which the user is involved in synchronizing the the units of computation. Cilk and Mentat requires the user to explicitly state the interaction and synchronization between the units of parallelism while this is not necessary in our runtime system. A more complete comparison of our work with other data-driven model can be found in [17, 18] which studied the locality and paralllelism of the runtime system. This paper describes the performance of a dependence-driven execution in multiprogrammed environment where there are more processes than processors.

We focus a little on the Chores runtime system which is most similar to our own work along the dimensions mentioned . In the Chores runtime system, a per-processor worker (a user-level thread) grabs chunks of work from a central queue using the guided self-scheduling method. Since Chores uses a guided self-scheduling to schedule loops, the system requires that a multi-

dimensional iteration space be collapsed into a single dimension. For example, to schedule the loop in a two dimensional wavefront application (as required in the mean value analysis), the Chores system linearizes the iteration space as space as shown in figure 10. In this example, the element (i, j) represents the computation of performance measures for a queuing network where i is a class 1 customer and j is a class 2 customer. Each element (i, j) can only be readied (enabled) when both $(i-1, j)$ and $(i, j-1)$ are completed. Initially, only $(0, 0)$ is ready for computation.

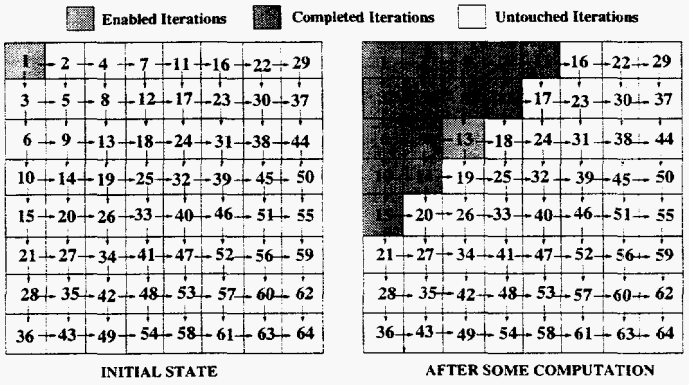


Figure 10: A linearization and dependencies of a two-dimensional space for MVA

As in our runtime system, new iterations are enabled as dependencies on them are satisfied. However, the set of enabled iterations must always be a contiguous range. This means that new iterations can only be enabled if these iterations extend either the upper bound or the lower bound of the range of enabled iterations; no holes can be created. The single contiguous range requirement is necessary to apply guided-self scheduling.

3 Experimental Results

To compare the effect of multiprogramming on various methods, we measured total execution time of three applications on a 16 processor SGI Origin under various loads: 1) on an idle machine 2) on a machine with 8 of the 16 processors used for a different job and 3) on a machine with different tasks using all 16 processors. These experiments correspond to a multiprogramming level of 1.0, 1.5, and 2.0, respectively. Ideally, we would expect that a multiprogramming level of 1.5 would cause the application to take 50% longer to execute and a multiprogramming level of 2.0 would cause it to take 100% longer with respect to the speed of the application running on an idle machine. However, because of the overhead associated with context switches, and loss of memory locality, this is often not the case. The applications consisted of Red/Black SOR (size 2048x2048), Multigrid solver (size 2048x2048), and the Mean Value Analysis. On an idle machine, MVA and Multigrid have a highly dynamic fluctuation in workload, whereas Red/Black SOR suffers little from load imbalances.

In the performance graphs shown in this section, we show, for each method, the absolute time elapsed to execute the application at the three levels of multi-programming. The darker shaded portion of each bar shows the 95% confidence interval. In some bars, the confidence intervals are not large enough to be seen.

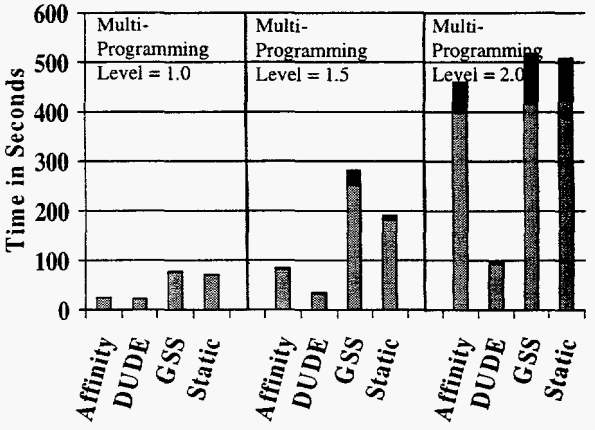


Figure 11: Execution Times for Red/Black SOR (size = 2048x2048)

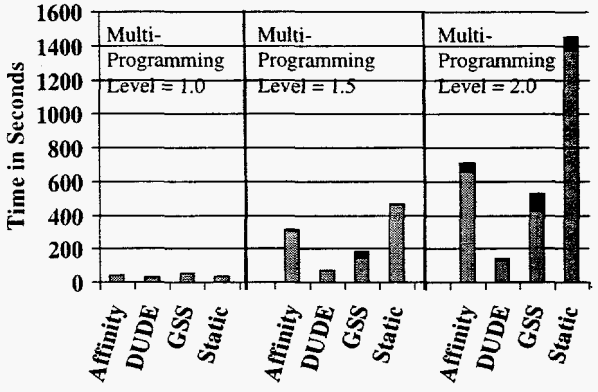


Figure 12: Execution Times for Multi-Grid (size = 2048x2048)

3.1 Red/Black SOR

Figure 11 shows the comparison of the performance of the Red/Black SOR (with matrix size of 2048x2048) using four methods: DUDE, Affinity scheduling, Static scheduling, and Guided-self scheduling. In the case of DUDE, we saw a 57% slowdown when running at a multiprogramming level of 1.5 and a slowdown of 270% at a multiprogramming level of 2.0. However, these slowdowns were very small in comparison to the slowdown observed in other methods. The Figure 11 also indicates that the 95% confidence interval for DUDE is much lower than the other methods.

3.2 Multi-Grid

Figure 12 compares the performance for the Multigrid Solver. A multiprogramming level of 1.5 caused DUDE to slowdown by 154%, whereas a level of 2.0 caused a slowdown of 374%. Again, the 95% confidence interval indicates that there is little variance when using DUDE.

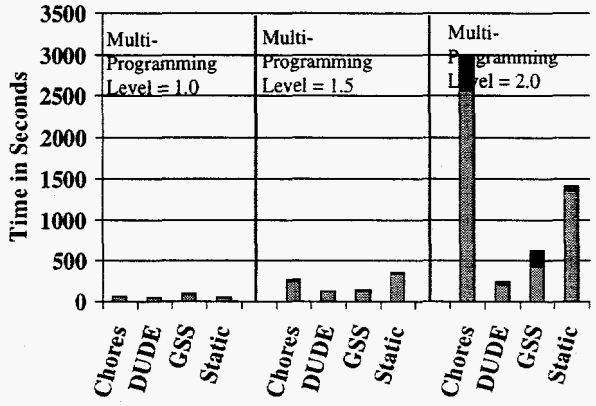


Figure 13: Execution Times for Mean Value Analysis

3.3 Mean Value Analysis

The performance results for this application are shown in figure 13. When at a multiprogramming level of 1.5, DUDE suffered a slowdown of 174% while a multiprogramming level of 2.0 caused a slowdown of 400%. These degradations in performance are small in comparison to the other methods.

3.4 Summary

Performance can severely degrade in a multiprogrammed environment. Applications that have static behavior when run on an idle machine can exhibit highly dynamic load imbalances when run on multiprogrammed environment. The conventional solution to these problems is to have the application interact with the operating system kernel. Since this requires an extension to the kernel, and requires applications to be aware of these extensions, these conventional solutions are difficult to use. In this section, we showed that it is possible to improve the performance of applications running on a multiprogrammed environment without using kernel extensions.

4 Conclusion

Performance can severely degrade in a multiprogrammed environment. Applications that have static behavior when run on an idle machine can exhibit highly dynamic load imbalances when run on multiprogrammed environment. The conventional solution to these problems is to have the application interact with the operating system kernel. Since this requires an extension to the kernel, and requires applications to be aware of these extensions, these conventional solutions are difficult to use. In this section, we showed that, by overlapping the computations of consecutive phases of a data parallel program, it is possible to improve the performance of applications running on a multiprogrammed environment without using kernel extensions.

References

[1] T.E. Anderson, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):52–79, February 1992.

[2] Derek L. Eager and John Zahorjan. Chores: Enhanced run-time support for shared memory parallel computing. *ACM. Trans on Computer Systems*, 11(1):1–32, February 1993.

[3] D. Gannon and J.K. Lee. Object oriented parallelism. In *Proceedings of 1991 Japan Society for Parallel Processing*, pages 13–23, 1991.

[4] A. S. Grimshaw. Easy to use object-oriented parallel programming with mentat. *IEEE Computer*, pages 39–51, May 1993.

[5] A. S. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic object-oriented parallel processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, pages 33–47, May 1993.

[6] Dirk Grunwald. A user's guide to a.w.e.s.i.m.e: An object oriented parallel programming and simulation system. Technical Report CU-CS-552-91, University of Colorado, Boulder, 1991.

[7] J. Gurd, C.C. Kirkham, and A.P.W. Boehm. The manchester prototype dataflow computer. *Communication of the ACM*, 28:34–52, January 1985.

[8] J. Gurd, C.C. Kirkham, and A.P.W. Boehm. *The Manchester Dataflow Computing System*, pages 516,517,519,520,529. North-Holland, 1987.

[9] High Performance Fortran Forum HPFF. Draft high performance fortran specificition, version 0.4. In *Proceedings of 1991 Japan Society for Parallel Processing*, page Available from anonymous ftp site titan.cs.rice.edu, 1992.

[10] L. Kontothanassis and R. Wisniewski. Using scheduler information to achieve optimal barrier synchronization performance. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, May 1993.

[11] L. Kontothanassis, R. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM TOCS*, 15(1), 1997.

[12] E.P Markatos and T. J. LeBlanc. Load Balancing vs Locality Management in Shared Memory Multiprocessors. In *Intl. Conference on Parallel Processing*, pages 258–257, St. Charles, Illinois, August 1992.

[13] Jose Moreira and Constantine Polychronopoulos. Autoscheduling in a shared memory multiprocessor. In *Proceedings of the IEEE/USP International Workshop on High Performance Computing Compilers and Tools for Parallel Processing*, March 1994.

[14] John Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Distributed Computing Systems*, pages 22–30, Oct 1990.

[15] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[16] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 159–166, Dec 1989.

[17] Suvas Vajracharya and Dirk Grunwald. Dependence-driven run-time system. In *Proceedings of Language and Compilers for Parallel Computing*, pages 168–176, 1996.

[18] Suvas Vajracharya and Dirk Grunwald. Loop reordering and pre-fetching at runtime. In *High Performance Networking and Computing*, November 1997.

[19] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.