# The DUDE Runtime System: An Object-Oriented Macro-Dataflow Approach To Integrated Task and Object Parallelism

Dirk Grunwald Suvas Vajracharya
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{grunwald,suvas}@cs.colorado.edu)

CU-CS-779-95 August 1995

University of Colorado at Boulder

Technical Report CU-CS-779-95
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Dirk Grunwald Suvas Vajracharya
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{grunwald,suvas}@cs.colorado.edu)

# The DUDE Runtime System: An Object-Oriented Macro-Dataflow Approach To Integrated Task and Object Parallelism

Dirk Grunwald Suvas Vajracharya
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{grunwald,suvas}@cs.colorado.edu)

August 1995

**Abstract**

Modern parallel programming languages allow programmers to specify parallelism using implicitly parallel constructs such as data parallel or object parallel methods, and explicitly parallel constructs, such as `doall`, `doacross`, `parallel_section` or programmer-level threads. In this paper, we present the design of a runtime system that executes data-parallel (or object-parallel) code in the presence of explicit parallelism. This facilitates load balancing between data-parallel computations running in threads of distinct parallel sections, as well as inter-loop load balancing. Although sufficient runtime structure is provided for most extant languages, the runtime system is extensible, allowing compilers to customize the runtime system.

To motivate why such a runtime system is desirable, we use show performance improvements for programs with complex data dependence relations, such as multigrid solvers.

## 1 Introduction

Most efforts on simplifying or improving parallel programs has focused either on large compiler systems, such as HPF Fortran, or small optimizations such as improved synchronization or barrier algorithms. A *runtime system* is the interface between a compiler and the underlying operating system and hardware; synchronization algorithms are one part of a runtime system. The design of runtime systems can dramatically affect the way compilers convert programs into a parallel form.

Scheduling is the central function of most runtime systems. Poor scheduling decisions can introduce significant performance problems in parallel programs. Runtime systems for "high performance" systems used for data-parallel languages usually provide a single virtual processor for each physical processor. These virtual processors then perform *loop-level* scheduling for the work of individual parallel constructs, such as `doall` loops. Other runtime systems support a large number of virtual processors, or threads, and concentrate on efficiently scheduling those threads [5]. Both loop-level and thread-level scheduling decisions require information about the machine architecture, taking into account the number of processors, memory bandwidth and communication delays.

Furthermore, such scheduling mechanisms should be portable across a range of architectures to simplify the code that must be generated by a compiler.

Increasingly, programming languages need support for a number of virtual threads while still providing an infrastructure for efficient loop-level scheduling. Virtual threads can be used to invoke multiple program sections in parallel and to mask communication latency for message passing programs. Loop-level scheduling is still needed for executing data-parallel operations, since the overhead of loop scheduling is usually significantly less than that for thread scheduling.

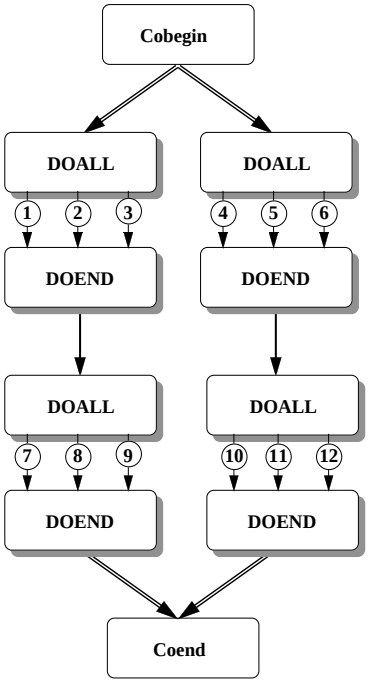
In this paper, we show how an integrated runtime system can be designed to perform both loop-level scheduling and task scheduling. Our runtime system is designed for shared memory computers that may be further connected using a message passing interface. We assume the shared memory computers have a pronounced memory hierarchy; examples of such architectures are the KSR-1 [21] and distributed shared memory systems [20]. Compilers must target a specific machine model supported by the runtime system, and we feel the art of designing a runtime system is to provide an interface with the most generality that can be implemented efficiently across a number of systems. More general constructs allow the compiler to defer scheduling decisions until execution time, when they can be optimized by the runtime system; however, this only works if the runtime system is efficient.

Our runtime system uses a *macro-dataflow* approach; the definition, or producer, of data and the use, or consumer, of that data are explicitly specified during execution. This distributes synchronization overhead and provides a very flexible scheduling construct. We call our runtime system the Definition-Use Description Environment, or DUDE, and it is currently implemented as a layer on top of the existing AWESIME threads library [15]. Normally, dataflow execution models have been associated with dataflow processors [2, 26, 26], but the macro-dataflow model has been implemented in software as well [3, 29]. Often, as in the case of MENTAT, an entire language is designed around the macro-dataflow approach.

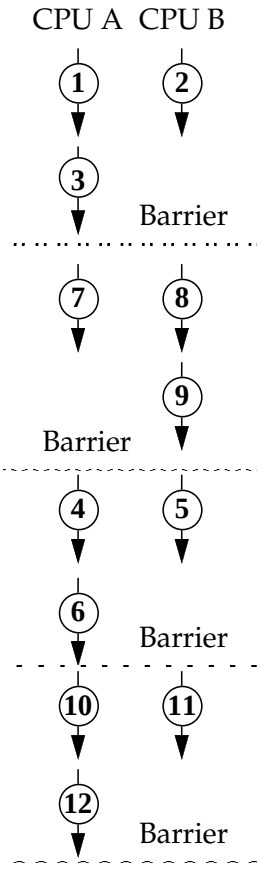
By comparison, we simply use the macro dataflow notions to provide a description of the dependence relations in a program. In many ways, the DUDE system is a fusion of existing macro-data flow techniques and thread and loop-level scheduling systems. We discuss these related systems in §4. For the moment, we describe a sample program implement in the DUDE environment and the performance we have measured.

### 1.1 Performance for the Quasi-Geostrophic Multigrid Application

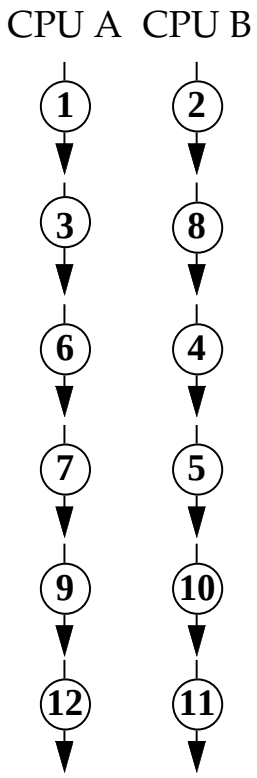
Figure 1 diagrammatically illustrates a program that specifies task parallelism using the `cobegin` construct and parallel iteration using the `doall` construct. This program illustrates one possible structure for the multigrid solver of a quasi-geostopic multigrid (QGMG) application, used by an NSF Grand Challenge project with which we are associated. We describe the problem in more detail later; for now, it suffices to see that we have two independent tasks, represented by either fork of the `cobegin` statement. Each branch performs two data-parallel operations, specified by `doall` operations. In a true data-parallel language, other constructs may replace the `doall` operations, but the semantics would be similar. There are several details of our program not shown by this diagram. Each pair of `doall` loops in the `cobegin` statement has a dependence distance of one. Thus, the iteration marked '1' must finish before the iteration marked '7' can begin; however, iteration '1' and '4' may execute concurrently.




(a) Diagrammatic Illustration of Program Combining Task and Data Parallelism




(b) Schedule for Conventional Runtime System




(c) Possible Schedule for Proposed Runtime System

**Figure 1:** Example Program and Schedules on Two Processor System

There is considerably more parallelism in this program than the `cobegin` and `doall` semantics imply. We assume the compiler may be able to determine some of this dependence information – computing dependence information has been extensively studied, and there has been recent work on analyzing explicitly parallel programs [12, 9]. A conventional runtime system might implement this program by closely following the structure of the original program. This is illustrated in Figure 1 for two processors. Each `doall` construct is executed in its entirety, and the execution of `doall` blocks is separated by barrier synchronization.

There have been numerous methods proposed to schedule the individual iterations of the `doall` loops, such as guided self scheduling and factoring [28, 17]. Conventional runtime systems use static, dynamic or some variant of adaptive scheduling to assign iterations to specific processors. Typically, a single dimension of a multidimensional iteration space is scheduled, although some researchers have considered scheduling nested loops [31]. Using a conventional runtime system, dependence constraints between iterations are enforced by event synchronization (`post` and `wait`) or by nesting sequential constructs within the outer parallel loop. Eager *et al* [10] proposed a scheduling paradigm, called *chores*, that is similar to loop-level scheduling of multi-dimensional iteration spaces. The Chore system directly represents multi-dimensional iteration spaces using run-time data structures. The iteration space is dynamically subdivided, essentially providing the same scheduling decisions as existing dynamic scheduling algorithms. However, dependence constraints within an iteration space can be specified by a dependence function. The DUDE runtime system inherits much of its structure from the Chore system; we extend chores to include inter-operation dependence constraints and and different mechanisms for implementing dependence functions.

A performance comparison of the conventional and the DUDE runtime systems for the two independent multigrid solvers having the structure described in Figure 1 is shown in Figure 2. The performance improvement shown in this figure can be attributed to two effects: the elimination of barrier synchronization and the scheduling of two tasks (each a multigrid solver) in parallel. The benefits of eliminating barrier synchronization alone can be seen in Figure 3, which is a comparison of conventional methods with the proposed runtime system for a single data-parallel task, the Red/Black SOR.

The rest of the paper is organized as follows. We start by explaining the QGMG application in §2 to be in the position to explain what language constructs are desirable and to motivate the design of the DUDE runtime system. This then sets the stage for §3, which describes our proposed runtime system. Section 4 surveys prior work and why there is a need for the proposed runtime system. In §5 we describe in detail the performance results fore-shadowed in this section. Finally, we close with discussion of future work and conclusions.

## 2 Sample Application: Quasi-Geostrophic Multigrid Solver

In this section we describe the Quasi Geostrophic Multigrid (QGMG) solver to motivate the design decisions of the proposed runtime system. The quasi-geostrophic equations describe the nonlinear dynamics of rotating, stably stratified fluids which is used to numerically simulate the highly turbulent nature of planetary flows of the Earth's atmosphere and ocean. Planetary-scale fluid motions in the Earth's atmosphere are important to the the study of Earth's climate. A more complete description of the QGMG application is available [7, 32]. Here, we concentrate on only the computational aspect of the problem as it relates to DUDE runtime system. As described [7],

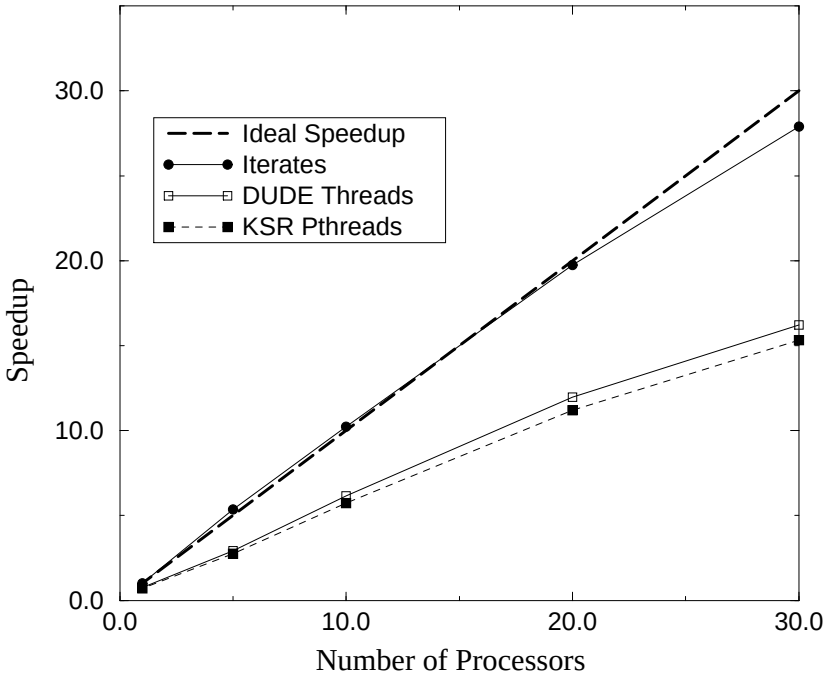



**Figure 2:** Speedup for Multigrid Solver on the KSR1.

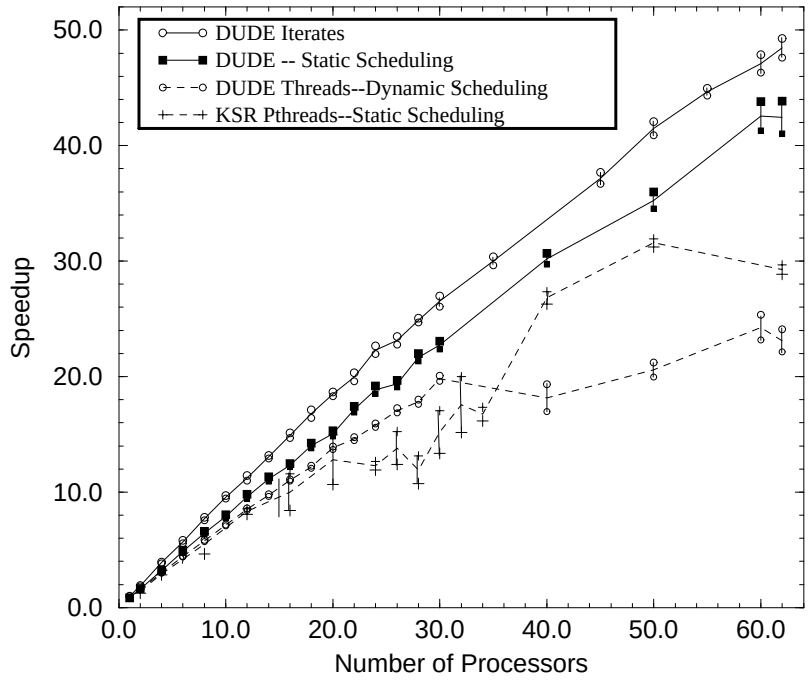



**Figure 3:** Speedup for Red/Black SOR on the KSR1

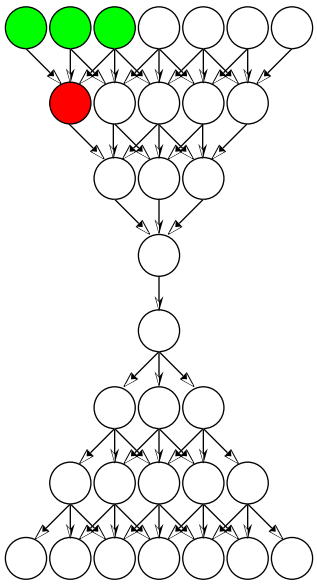


**Figure 4:** Dependence relations in a 1-Dimensional multigrid application during a single V-cycle

the QG equations of motion are

$$\frac{\partial q}{\partial t} + \frac{\partial \psi}{\partial x}\frac{\partial q}{\partial y} - \frac{\partial \psi}{\partial x} + \beta\frac{\partial \psi}{\partial x} = -D$$

where

$$q = \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial}{\partial z}\left(\frac{1}{S(z)}\frac{\partial \psi}{\partial z}\right)$$

The best method for solving these equations is the multigrid solver. The two main variables in these equations are $q$ and $\psi$, each of which requires a multigrid Solver and its associated data structures. In a language that supports `cobegin`, two tasks can be spawned to independently solve for the two variables. Each task executes data-parallel operations, as shown earlier in Figure 1, with additional interaction between the two tasks. The combined data and task parallelism provides many opportunities for improving performance.

The multigrid method has received much attention due to its fast convergence and the challenge of parallelizing the algorithm [6, 30, 8, 13]. The basic idea of the multigrid method is to obtain an initial approximation for a given grid by first solving on a coarser grid. Since the coarser grid is simple to compute, we can use a iterative method such as the Gauss Seidel method to get an approximation solution on the coarse grid and then interpolate this approximation to the finer grid. A recursive use of this basic idea leads to the full multigrid algorithm: First use relaxation (smoothing) on an input matrix to obtain an approximation with the smooth error. Using this error, a correction to this approximation is computed on a coarser grid. Computation is mapped to a coarser grid using a restriction operator, the simplest of which is to simply copy some of the points from the fine grid onto the coarse grid. The coarsest grid can be solved exactly, after which we begin interpolating (prolonging) the correction back to finer grids.

The algorithm we described uses a V-cycle but there are many variants. Restricting and prolonging amounts to climbing up and down a pyramid of matrices where the base is the finest grid

and the coarsest grid is the top of the pyramid. It is easier to understand the dependence constraints using a simpler one-dimensional multigrid solver as an example. Figure 4 shows the dependence relations between levels of a V-cycle for a one-dimensional problem. Each circle represents the execution of a single iteration of the relaxation function. Each level of the pyramid consists of three operations: Smooth the even elements of the matrix, smooth the odd elements, obtain an approximation and restrict to next coarsest grid level if going up the pyramid or prolong to next finest grid level if going down. There is a dependence across the levels of the pyramids, as indicated by the arrows in Figure 4. Normally, the dependence relations shown in Figure 4 are satisfied by completing all iterations in each level before starting the next level. Figure 4 shows that this is not necessary; the iteration indicated by the lower darker circle can be started when the three iterations on which it depends finish.

Conventional parallel implementation of the multigrid method involves partitioning the finest grid matrix among the available processors. Processor utilization is acceptable on fine grids, but as the algorithm climbs up the pyramid to coarser grids, the majority of the processors will be idle. Recall that the QGMG problem must solve two multigrid problems; these can be executed concurrently, but many conventional runtime systems do not support the concurrent computation of two data parallel computations. A third problem is that barrier synchronization strictly forces an operation to complete before the next one begins. The operations described above (smooth, prolong, restrict) are only dependent on neighboring elements to complete, not the entire matrix. If we allow some processes to continue processing the next operation immediately after the neighboring points have been calculated, we can greatly improve processor utilization.

The DUDE runtime system addresses all of these problems. We eliminate the barriers implicit in the multigrid algorithm, substituting nested explicit dependence information. The dependence information indicates when higher levels of the V-cycle can start execution, and multiple parallel operations can be executed in parallel.

Data dependence constraints between groups of iterations, termed *iterates* in our system, are enforced by runtime dependence information determined during compilation. Control dependence operations, such as the barriers in the previous examples, are also controlled by runtime representations. In our current implementation of the runtime system, this example program runs without using barrier operations. All dependence information is specified using "precedence edges" in the runtime structures. Since this precedence information only involves a subset of the processors in the system, synchronization is faster, reducing overhead. In effect, the DUDE runtime system provides a concrete runtime implementation for the dependence information shown in Figure 4.

# 3 The Design of DUDE

The DUDE runtime system is based on AWESIME [15] (A Widely Extensible Simulation Environment), an existing object-oriented runtime system for shared-address parallel architectures. The AWESIME library currently runs on workstations using Digital Alpha AXP, SPARC, Intel 'x86', MIPS R3000 and Motorola 68K processors, as well as the KSR-1 massively parallel processor. The AWESIME library has been in use for a number of years, primarily for efficient process-oriented discrete event simulation – for example, Tera Computer Corporation uses AWESIME for operating system simulations.

We have extended the AWESIME run-time system to implement the Definition-Use Description Environment. In DUDE, objects of class `Thread` are a basic unit of task parallelism and objects

of class `Iterate` are a basic unit of data parallelism. Both `Thread` and `Iterate` are subclasses of the `PObject` (parallel object) class, which represents any unit of parallelism managed by the scheduler. A `Thread` has a stack and state information. As with many runtime systems, the overhead of saving this state information during context-switches can be minimized by creating only one `Thread` per processor, but programmers are able to create any number of threads. In related work, we are using whole-program compiler optimization to reduce the space and time overhead for threads. Precedence constraints due to data dependences in the application program can be satisfied for `Thread`s using the synchronization mechanisms supported by DUDE, such as *barriers* or *semaphores*. These operations only make sense for stateful concurrent objects that can block and resume execution (i.e., threads). By comparison, `Iterate`s run to completion and are not context switched. `Iterate`s can not block on barriers or semaphores, since they have no state; instead, explicit precedence information is used. Because `Iterate`s lack state, they can be created and managed much more efficiently than threads.

The abstraction to `PObject` over these two subclasses allows applications to use both `Thread` and `Iterate` objects. A `Thread` or `Iterate` can only be created by sub-classing the existing classes. For example, an iterate describing a particular computation would be represented by a subclass of `Iterate`. All behavior specific to that computation will be encapsulated in the subclass. In this paper, we frequently refer to the activity of a `Thread` or `Iterate`, but such references should be understood to refer to a subclass of those classes.

As with all objects in C++, the class constructor is invoked when an iterate or thread is created. Arguments to the iterate or thread are specified in the application program and are recorded in the corresponding instance variables. Any `PObject` can be bound to specific processors using the `CPUaffinity` method. The `PObject` class provides a *virtual function*, `main` to customize the activity of each thread or iterate. The `main` method is the starting point for a new `Thread` or `Iterate` and is provided by subclasses of `Thread` and `Iterate`. Thus, the body of `main` can be a unit of execution in a data parallel loop or the body of a task.

Parallel objects are scheduled using a `CpuMux`, or CPU multiplexor. There are several subclasses to the `CpuMux` base class, defining the scheduling policy to be used for specific application. Each CPU multiplexor repeatedly selects a `PObject` to execute, and executes that object. The execute method specialized for `Thread`s will context switch at this point, while an `Iterate` will directly execute the function associated with the individual `Iterate` object.

Dynamic dispatch based on object type is used through-out AWESIME and DUDE. The `CpuMux` object represents a hardware processor. Using the object-oriented model provided by C++, we provide specialized `CpuMux` subclasses for different parallel architectures that provide different work-sharing strategies. The most common work-sharing mechanism uses a separate scheduler for each `CpuMux`, and `CpuMux`'s "steal" from each other if they are idle. As another example of dynamic dispatch, users can select a barrier algorithm that is most appropriate to the architecture [16] or problem.

The DUDE runtime system uses the abstraction and inheritance constructs of C++ to keep the scheduling policy, the underlying hardware, the type of objects being scheduling, the type of synchronization and other aspects of the system mutually orthogonal. As we will see, we need not sacrifice efficiency for this generality and modularity. Dynamic dispatch is also the basis of loop scheduling using the `Iterate` class, which we describe in some detail.

**Figure 5**: The Iterate Class

### 3.1 Data Parallelism: Computation using Iterates

The `Iterate` class is the core construct for data parallel computation in DUDE. The `Iterate` class provides a mechanism that can best be described as a large grain dataflow execution model. The goal is to relieve the application programmer or the compiler from concerns regarding locality of data, enforcement of synchronization of data constraints and scheduling.

Figure 5 shows the instance variables and the methods that the application programmer must specify. The `main` method is the operation that is to be performed on the data. The descriptor specifies a portion of the parallel loop accessed by the `main` method. The lower bound, upper bound and the stride can all be extracted from the descriptor. Each `Iterate` also contains an internal loop control variable and a loop terminating variable. All of these variables are initialized in the `Iterate`'s constructor (called `Iterate101` in the diagram).

The remaining methods are used to determine the continuation of an `Iterate`. When an `Iterate` finishes execution, the scheduler determines if the completed `Iterate` has satisfied any precedence constraint. Figure 7 shows the scheduling engine. The scheduler calls the `generateDescriptors` method of the completed `Iterate` which returns a list of data descriptors. Each descriptor represents an arc in the precedence graph. This descriptor is then used as a key to a table that counts of number of `Iterate`s that have finished and the number needed to satisfy the dependence constraint. Constraints are satisfied if the count is equal to the expected value of the dependence count. The `getNumDeps` method returns the number of expected dependents. If the constraints are satisfied, then the `makeContinuation` method is used to instantiate the continuation and add them to the work heap. The runtime system performs all the synchronization required to insure that the precedence constraints are satisfied. The application programmer or the compiler need only express the dependence information in the form of the `generateDescriptor` method.

Note that the dependence constraints also distribute the synchronization that occurs in the program. In distributed shared memory computers, such as the KSR-1, synchronization among a large number of processors causes particular cache lines to become hot-spots [27]. By distributing the activity over a number of synchronization variables, the hardware parallelism supported by the multiple communication levels in a system such as the KSR can be exploited.

By providing the concept of dependence and use specification in the runtime system, we can also execute multiple parallel operations concurrently. The QGMG program must solve two multigrid problems to advance a single time-step. A traditional runtime system, or even advanced systems such as the Chores model [10] must sequentially schedule the computation in each `doall` or loop nesting. By allowing all operations to be evaluated in parallel, we increase the scheduling opportunities, allowing the runtime system to select a better schedule.

The iteration space is initially sub-divided into fixed sized chunks, with each chunk being represented by an `Iterate` object. The iteration space is described using a symbolic representation, much

```
void RedSOR::main()
{
   for (short i = getSY(); i <= getEY(); i += getST()) {
     for (short j = getSX(); j <= getEX(); j += getST()) {
       mydata[i][j] = Func(mydata[i-1][j] + mydata[i][j+1]
                           + mydata[i+1][j] + mydata[i][j-1]));
       mydata[i+1][j+1] = Func(mydata[i][j+1] + mydata[i+1][j+2]
                               + mydata[i+2][j+1] + mydata[i+1][j]));
       }
     }
   }
}

int RedSOR::getNumDeps()
{
  return 5;
}

BlackSOR *RedSOR::makeContinuation(DESC desc)
{
  return BlackSOR::MyAlloc(desc.J,desc.K);
}

DESC * RedSOR::generateDescriptors()
{
    // get current index to this Iterate.
    int J = getJ();
    int K = getK();
    if (getLoopIndex() == getEnd()) return NULL;
    // create dependence vector
    DESC *desc = FormDescriptor(J,K, -1,0, +1,0, 0,-1, 0,+1);
    return desc;
}
```

**Figure 6**: Some Methods from the Red/Black SOR Iterate

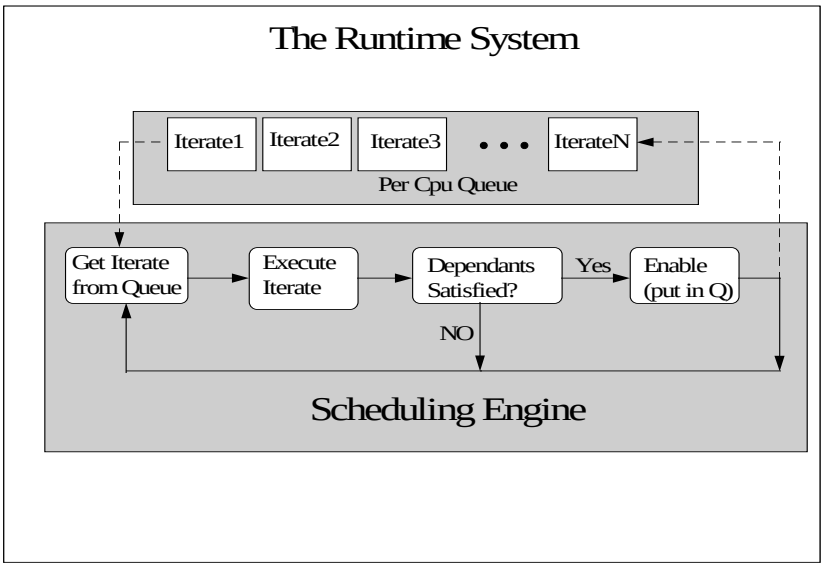



**Figure 7:** Scheduling Engine of the Proposed Runtime System

as was done in the Chores system; however, we found that the repeated evaluation of the symbolic dependence constraint was too slow. Instead, the first time a construct is executed, the symbolic representation is used to create a series of `Iterate` instances that represent a sequential execution of a subset of the iteration space. These sequential sections are then dynamically scheduled across multiple processors. The decomposition of the iteration space can be cached to reduce the time to start the computation if that routine is executed repeatedly.

One of several distribution techniques, such as block or distribution, may be used to initiate the decomposition. If the processors experience load imbalance, as determined by a scheduling heuristic, these fixed sized `Iterate`s may be further subdivided during the execution of a parallel construct. When all the subdivided chunks are completed in that iteration, the original `Iterate` that was subdivided resumes its initial size for successive executions. Partitioning need not be concerned with the data dependence specified in the `Iterate` since the partitions are only in effect for the duration of one loop iteration. In other words, completion of any one of the subdivided parts is not sufficient to begin enabling continuations; the entire portion must be completed. This reduces the overhead of subdividing the computation, because the dependence information of continuations does not need to be modified.

The rational for initially decomposing an *Iterate* into fixed sized parts is threefold. First, fixed size chunks simplify maintaining the dependence information, and make that process more efficient. Allowing variable size chunks imply a less efficient data descriptor that take a range of values instead of indices. We initially implemented such a structure, similar to the Data Access Descriptor [4], but found it was too slow in practice. Secondly, and more importantly, fixed size chunks allows the scheduler to establish an affinity between a chunk and the processor improving data locality. Each chunk has a preferred processor when it is rescheduled on the next iteration of the loop. This affinity is only compromised if there is a great load-imbalance or there is insufficient work left to be done. Thirdly, contention for a single large chunk at the beginning of the computation is avoided because each CPU can start with an `Iterate` from its own local queue.

Initially these chunks or `Iterate`s are distributed to local queues of the `CpuMux`'s. The `CpuMux` for each individual processor grabs an `Iterate` from the local queue to process. If this queue is empty, it attempts to steal work from another `CpuMux`. When the total number of `Iterate`s to schedule is below a certain threshold, the `CpuMux` divides an `Iterate`, removing it from the queue only when all the partitions have completed. Upon completion of an `Iterate`, the scheduler marks that object with its processor number. This information will be used in the next iteration to decide which local queue should be preferred for this `Iterate`.

Iterates are created as the program executes and encounters parallel constructs. For example, the execution of a `doall` corresponds to the creation and scheduling of a collection of `Iterate`s. Threads wait for a specific parallel construct to complete by blocking on a semaphore, and the continuation for the `Iterate` representing a `doall` releases that semaphore. The main program is represented by a `Thread`, and can create additional threads or iterates as needed. In fact, iterates can be used to create threads; this will be used to schedule threads to mask the latency of message passing applications.

# 4 Prior and Related Work

There are a number of existing runtime systems for parallel computers. Why should we build yet another runtime system? First, many existing runtime systems provide a limited set of abstractions

for concurrency. For example, the PTHREADS [25] and PRESTO [5] libraries only provide threads and traditional synchronization mechanisms such as barriers and locks. We feel these abstractions are too simple to allow a compiler to generate code that can efficiently manage the resources of a complex architecture, such as a distributed shared memory computer.

More recently, there have been a number of programming languages and runtime libraries stressing a diversity of parallelism constructs. The Chare language [18], and more recently the CHARM++ [19] system, use actor-like message semantics and a continuation computation model. In the Charm runtime system, used by the Chare language, tasks are spawned by sending initiation messages to processors. Tasks subdivide work by creating other tasks. The Chare system was originally designed to support parallel logic programming languages such at ROLOG, but has evolved into a more general programming tool. The charm system does not implement 'threads' in the conventional sense - state can not be saved and resumed at a later time. Furthermore, the runtime semantics are targeted towards message passing environments, and does not provide explicit support for shared address environments.

The Chores [10] and Filaments [11] systems are the most germane runtime systems we've encountered. Filaments [11] are extremely fine grain stateless threads consisting of a pointer to code and a list of arguments. Engler *et al* showed that filaments, due to their low overhead ($\approx$ 16 bytes each), were suitable for exploiting fine grain parallelism on a variety of programs. For example, in the Red-Black SOR problem, each point in the matrix would be represented by a filament. This extremely fine granularity permits efficient load balancing, but has certain drawbacks. First, conventional compiler optimizations such as strength reduction, induction variable detection and common subexpression elimination can not be performed because the higher level looping is used to *create* filaments. Thus, it is difficult for each filament to take advantage of the state of the parent process; this results in good speedup, but poor performance. Furthermore, the overhead of representing filaments is considerable for large problems. Our application group wants to process matrices containing $1024^3$ elements - in the filaments model, the programmer (or compiler using the Filaments runtime system) decomposes the problem prior to execution. Thus, for fine-grained load balancing, we might break the problem into $1024^2$ components — at a cost of 16Mbytes of memory and an outer loop that creates and initializes one million filaments.

The Filament system provides three types of threads: barrier filaments, run-to-completion filaments, and fork-join filaments. DUDE presently provides two types of parallel objects: blocking threads with stacks and `Iterate`s which roughly correspond to a combination of barrier and run-to-completion Filaments. The difference between Filaments and `Iterate`s are that `Iterate`s do not force a barrier synchronization. Instead, the system uses data dependence information to generate continuations when an `Iterate` completes, as described earlier. Also, `Iterate`s are created once and rescheduled multiple times. The advantage of this is not only the reduction of time to create and destroy the objects but, it allows the scheduler to establish an affinity of an `Iterate` to a particular processor.

As mentioned above, Filaments have the problem of overly fine granularity. The Chore system eliminates the problems by allowing 'atoms' (a sequential unit of work) to be aggregated and split dynamically. The Chores system is an extension of the work heap model in which one worker per processor grabs work from a queue. It extends the work heaps model by providing a description of the iteration spaces of multiply-nested loops and a symbolic specification for data dependence. The Chore systems also dynamically *partitions* portions of the iteration space if they are partitionable

(i.e. not atoms). The Chore model provides an implicit barrier when all the portions of a chore have finished

The DUDE `Iterate` class is also based on the idea of work heaps, as in Chores. However, DUDE allows the concurrent scheduling of multiple loops, supporting explicit parallelism while providing better scheduling opportunities. The Chores system can only schedule loops containing a single function. In many applications, loops contain multiple operations where each operation must wait for the completion of the previous operation(s). Traditional runtime libraries using threads simply insert a barrier synchronization between any two operations. The Chores system can eliminate barriers only if the operations between the barriers are the same, as in Gaussian elimination. We have extended this to eliminate barriers even when the barriers separate disparate operations. For example, in the multigrid solver, one must first smooth the even elements, then the odd elements, approximate, and finally restrict to the next level. The Chore system was designed with little regard for locality. Eager *et al* showed a improved performance due to a good load-balance on the Sequent Symmetry, an architecture that did not penalize severely for lack of data locality. The trend in more recent multiprocessor, such as the KSR-1, has been to increase the ratio of processor speed to communication speed, making good locality as important as good load-balance [24]. The DUDE runtime system allows an affinity between `PObject`s and processors.

Finally, the Chores system was intended to be used also by application programmers - to simplify programming, chores have the *option* of suspending execution, but the normal convention is that individual chores usually run to completion. If a chore blocks, a *scheduler activation* creates a new 'worker thread' to continue chore execution. By comparison, we are providing a library for an object-parallel language, and assume the compiler can distinguish between blocking and non-blocking operations. This simplifies the runtime system design and makes it more portable.

Graham *et al* [14] extended the design of an existing compiler to overlap the execution of different loop constructs. Thus, the `doall` operations in each thread may be combined, reducing the execution overhead and improving load balance. Our proposed runtime combines elements of the Chores system and the optimizations proposed by Graham *et al.* We extend these prior systems by combining the flexible *intra*-operation scheduling of the Chores system while the reducing the constraints of *inter*-operation scheduling, such as barrier operations.

# 5 Performance Results

In this section we describe the performance of the Red/Black SOR and the Multigrid solver on the KSR1, a parallel cache-only memory architecture [21].

## 5.1 Red Black SOR Problem

The *successive over-relaxation* (SOR) method is an iterative technique to solve a linear system of equations. A common implementation of this technique uses the "five point stencil" to compute the `(k+1)`st iteration from the `k`th iteration by traversing the SOR iteration array in row major order. The Red-Black SOR algorithm [22, 23, 1] provides parallelism by dividing the mesh into "even" and "odd" meshes, like the red and black squares of a checkerboard. All even or odd elements can be processed concurrently.

We use the Red-Black SOR algorithm for two reasons. First, Red-Black SOR does not suffer from load-imbalance. As shown in the first method of Figure 6, the body of the loop to be exe-

cuted does not contain any conditional statements. This implies that if static scheduling is used, processors would arrive at the barriers at approximately the same time and therefore need not idle waiting for each other. Thus, if we can show that the DUDE runtime system can improve upon the static scheduling in this application, then we can show that the performance improvement would be greater on applications that suffer from load-imbalance in the body of the `doall` loops. Secondly, the Red-Black SOR algorithm is a kernel in the QGMG application, and the performance of this kernel is important for that application.

Figure 3 compares the performance of the Red-Black SOR algorithm running on a KSR-1 parallel computer using four methods: DUDE Threads with static scheduling, DUDE `Iterate`s, DUDE Threads using dynamic block scheduling and the KSR PTHREADS package with static scheduling. The results shown in Figure 3 are for a 1000x1000 matrix. For each method, the graph shows the average speedup and the and the 95% confidence intervals for that data point.

There is a one-to-one mapping between the KSR-PTHREADS and processors. We used the PTHREAD `bind` function to prevent migration of threads during the entire execution. Each thread is statically assigned $\frac{N}{P}$ rows of the input matrix. The PTHREADS barriers are used to insure that all the red computation are completed before any black computation is started (and vice versa). The DUDE threads used exactly the same scheduling as the KSR PTHREADS model. In both the thread methods, dividing the work into blocks instead of rows does not improve locality because the whole matrix must be traversed before the data is re-accessed; all the red computation must complete and synchronize at the barrier before any element is re-accessed by black computation. The improved performance is largely due to a more efficient barrier [16] which takes the hierarchical interconnect topology of the KSR1 into account. With dynamically scheduled DUDE threads, threads grab a block of the matrix from a global descriptor containing information on what work remains to be done. While data or work is not bound to a particular thread, the threads themselves are bound to processors as in the previous two methods. Consequently, there is a better load balance in this method at the expense of reference locality. Another disadvantage is that processors must contend for access to the global descriptor.

For the `Iterate`s implmentation, the matrix is broken into blocks of data, each of which is the responsibility of an `Iterate` object. The granularity is much smaller than in the case of either DUDE Threads or KSR PTHREADS. The main difference between this method and the dynamic thread method is that completed blocks can enable new blocks, overlapping the computation of the Red and Black computations. Locality is improved because Black operations may begin immediately after a Red operation if the precedence constraints are satisfied. This re-accesses the data for that region of the matrix before it has left the processors cache. Furthermore, each block may be further partitioned when the amount of work left is running low.

This experiment shows that dynamic scheduling can be detrimental on large shared memory multiprocessors. Notice that the performance of the dynamically scheduled computation becomes dramatically worse when more than 32 processors are used. The KSR-1 is structured as rings containing 32-processors; communication between rings is more expensive than communication within a ring. Thus, the improved load balance of dynamic scheduling comes at the cost of increased synchronization and communication overhead. It also shows that the efficiency of the native threads library, KSR-PTHREADS, can be less that than of a light-weight non-preemptive thread library. Lastly, it shows that the `Iterate` construct is even more efficient than the light-weight thread library. Both the `Iterate` and thread programs have a linear speedup, indicating they both scale well, but that the the `Iterate` method has lower scheduling and work-sharing overhead.

### 5.2 Multigrid Solver

Figure 2 shows the performance of combined task and data parallelism that occurs when traversing two independent loops, each solving a 1024x1024 matrix using the multigrid solver. Each loop independently solves a matrix size of 1024x1024. The speedup for the `Iterates` method is superior to the `Thread` methods.

In the thread method the input matrix is divided into an equal number of rows among the threads which are bound to physical processors. A barrier synchronization is used between each of the operations: smooth even, smooth odd, approximate, relax/prolong and the next level operations. Due to the halving of matrix dimension at the next highest level, the number of processors participating is reduced when climbing up the pyramid. Non-participating processors simple idle at the higher levels. The performance graph shows poor speedup for both the native and DUDE thread packages.

By comparison, the `Iterates` implementation solves both multigrid problems concurrently, and can solve the Red-Black SOR problem in each relaxation step using the method described in the previous experiment. Although this provides better scheduling opportunities during execution, we recognize that the near-linear performance does not scale indefinitely, although it does scale to a larger number of processors. To achieve both task and data parallelism, two threads are created. Each thread starts a multigrid solver using the data parallelism offered by `Iterates`. An `Iterate` class is created for each of the 5 operations: smooth the even elements, smooth the odd elements , approximate, prolong, and restrict. Initially only the SmoothEven `Iterate` is created and added to the work queue. As these complete, and the precedence constraint are satisfied, SmoothOdd `Iterates` (as specified in the `makeContinuation` method of the SmoothEven `Iterate` class) are enabled. The later operations execute in a similar fashion. After the Approximate `Iterate` completes, a choice of either enabling the Restrict `Iterate` or the Prolong `Iterate` is made in the `makeContinuation` method of the Approximate `Iterate`. Figure 2 shows that Multigrid solver using `Iterates` achieves super-linear speedup due to locality for small number of processors and near linear speedup for higher number of processors.

This experiment shows that the overlapped computation at each level of the multigrid computation, and the opportunity to overlap the execution of different multigrid planes results in improved speedup. This occurs both because of increased scheduling opportunities but also because of improved data locality and reduced scheduling overhead.

## 6 Conclusions

We have described an extensible runtime system for shared address architectures that supports both task and data (or object) parallelism. Our current implementation allows applications to specify precedence constraints between tasks and between different data parallel computations. Preliminary results show that for a data parallel application, we achieve better performance using runtime representations of control and data dependence than by using conventional thread decompositions. We are currently integrating the DUDE runtime system with the `pC++` object-parallel language, as part of a ARPA contract to develop a high-performance C++ infrastructure.

At the same time, our runtime system supports `Threads`, so we can express task or control parallelism between different sections of code that can execute in parallel. This is particularly important for "coupled" problems where we may be modeling two systems (structures & fluids,

oceans & weather) concurrently. Combined thread and object parallelism is important in programs such as adaptive mesh refinement, where data parallel operations are performed over a number of different arrays.

One feature not stress in this paper is that the DUDE runtime system is designed to be *extensible*, allowing the customization of scheduling policies and the introduction of new work-sharing structures. As parallel architectures are used for increasingly complex problems, extensible runtime systems that exploit additional degrees of parallelism within programs will be needed. We believe that this paper demonstrates that the object-oriented runtime systems offer excellent performance, and allow a great degree of extensibility.

### 6.1 Future Work: Profile-Driven Dynamic Scheduling Policies

Using information from profiling the application program, it is possible to determine the runtime behavior of programs and determine which scheduling policy is best suited for maximum load balance and parallelism in different sections of the program. For example, initialization of the elements of a huge matrix can be done in a *statically* scheduled parallel loop. A section of the program that has varying size code in different parallel `Thread`s based on inputs to the program may perform best with an adaptive scheduling policy. Thus, for better load-balance and parallelism, it may be worthwhile to change the scheduling policy dynamically as the program executes. We are extending the DUDE runtime system to support dynamically changing scheduling policies, by customizing the scheduling function based on profiling information from the application program.

## Acknowledgements

We are thankful to Harini Srinivasan for her comments on early version of this paper. We are also grateful to Clive Baille of University of Colorado for taking the time to explain the QGMG application to us. This work was funded in part by NSF grant No. ASC-9217394, ARPA contract ARMY DABT63-94-C-0029 and an equipment grant from Digital Equipment Corporation.

## References

[1] Adams L. M, Jordan H. F. Is SOR Color Blind? *SIAM Sci. Stat. Computation*, 7(2):490–506, 1986.

[2] T. Agerwala and Arvind. Data flow systems. *IEEE Computer*, 15(2):10–13, Feb 1982.

[3] Robert Babb. Parallel processing with large-grain data flow techniques. *IEEE Computer*, ??(??):55–61, July 1984.

[4] Vasanth Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9:151–170, 1990.

[5] B.N. Bershad, E.D. Lazowska, and H.M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software Practice and Experience*, 18(8):713–732, August 1988.

[6] A. Brandt. *Eliptic problem solvers.* Academic Press, New York, NY, 1981.

[7] Jeffrey B. Weiss Clive F. Baillie, James C. McWilliams and Irad Yavneh. Parallel superconvergent multigrid. In *submitted to Supercomputing '95.*, 1995.

[8] D. Gannon and J. van Rosendale. . *J. Parallel Distributed Computing*, 3:106–135, 1986.

[9] Dirk Grunwald and Harini Srinivasan. Data Flow Equations for Explicitly Parallel Programs. In *Conf. Record ACM Symp. Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

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

[11] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Efficient support for fine-grain parallelism. TR 93-13, Univ. Arizona, April 1993.

[12] Jeanne Ferrante, Dirk Grunwald, and Harini Srinivasan. Array Section Analysis for Control Parallel Programs. Technical Report CU-CS-684-93, University of Colorado at Boulder., November 1993.

[13] P. Frederickson and O. McBryan. Parallel superconvergent multigrid. In *Proceedings of the Third Copper Muntain Conference on Multigird Methods.* Marcel Dekker, 1989.

[14] Susan Graham, Steven Lucco, and Oliver Sharp. Orchestrating interactions among parallel computations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, April 1993. ACM, ACM.

[15] Dirk Grunwald. A users guide to awesime: An object oriented parallel programming and simulation system. Technical Report CU-CS-552-91, University of Colorado, Boulder, 1991.

[16] Dirk Grunwald and Suvas Vajracharya. Efficient barriers for distributed shared memory computers. In *Intl. Parallel Processing Symposium.* IEEE, IEEE Computer Society, April 1994. (to appear).

[17] S. F. Hummel, Edith Schonberg, and L. E. Flynn. Factoring, a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[18] L. V. Kalé and W. Shu. The Chare-Kernel language for parallel programming: A perspective. Technical Report UIUCDCS-R-89-1451, Univ. of Illinois, Urbana-Champaign, May 1989.

[19] Laxmikant Kale and Sanjeev Krishnan. Charm++: A portable concurrent object-oriented system based on c++. Technical report, Univ. of Illinois, Urbana-Champaign, March 1993.

[20] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Winter Usenix Conference*, 1994.

[21] Kendall Square Research, Boston, MA. *The KSR-1 System Architecture Manual*, April 1992.

[22] Adams L. M. *Iterative Methods for Solving Partial Differential Equations of Elliptic Type.* PhD thesis, Harvard University, Cambridge, Mass., 1950.

[23] Adams L. M. *Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers.* PhD thesis, Univ. of Virginia, Charlottesville, 1982.

[24] Markatos, E. P 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.

[25] Frank Mueller. *Pthreads Library Interface.* Florida State University, July 1993.

[26] Gregory M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* MIT Press, Cambridge, MA, 02141, 1991. (1988 MIT Ph.D. Thesis, also published as MIT LCS TR 432).

[27] G. Pfister and V. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[28] C. D. Polochronopoulous and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[29] Shankar Ramaswamy and Prithviraj Banerjee. Processor allocation and scheduling of macro dataflow graphs on distributed memory multicomputers by the paradigm compiler. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II-Software, pages II–134–II–138. CRC Press, August 1993.

[30] S.N. Gupta, M. Zubair and C.E. Grosch. . *J. of Scientific Commput.*, 7:263–279, 1992.

[31] P. Tang and P.C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. Int. Conf. on Parallel Processing*, pages 528–535. IEEE, August 1986.

[32] Irad Yavneh and James C. McWilliams. Multigrid solution of stably stratified flows: the quasi-geostrophic equations. In *submitted to J. Sci. Comp.*, 1995.