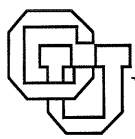
Application of an Object-Oriented Parallel Run-Time System to a Grant Challenge 3d Multi-Grid Code

Clive Baillie
Dirk Grunwald
Suvas Vajracharya

CU-CS-780-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Application of an Object-Oriented Parallel Run-Time System to a Grand Challenge 3d Multi-Grid code

Clive Baillie Dirk Grunwald Suvas Vajracharya
Department of Computer Science
Campus Box 430, University of Colorado
Boulder, CO 80309-0430

Abstract

We have taken a Grand Challenge 3d Multi-Grid code, QGMG, initially developed on the Cray C-90 and subsequently parallelized for MPPs, and implemented it using the DUDE *object-oriented, runtime system which combines both task and data parallelism. The QGMG code is a challenging application for two reasons. Firstly, as in all multigrid solvers, the most straightforward implementation requires that most of the processors idle at barrier synchronizations. Secondly, the QGMG code is an example of an application that requires both task and data parallelism: two multigrids (task parallelism) must be solved and each multigrid solver contains data parallelism. To address these challenges,* DUDE *loosens the requirement that all processes must wait at barriers, and provides integrated task parallelism and data parallelism. In this paper we describe the QGMG code and the* DUDE *object-oriented, runtime system in detail, explaining how we parallelized this Grand Challenge application.*

1 Introduction

We are associated with an NSF "Grand Challenge" application group modeling turbulence in a variety of media. One program, the QGMG (Quasi-Geostrophic Multi-Grid) application, currently achieves 6 Gflops on a Cray C90. We are using this code as one of our reference problems for improving runtime system performance.

Over the last year the QGMG code was implemented it in a portable parallel way on most of todays MPPs (Massively Parallel Processors) [1]. This was done via domain decomposition and message passing using PVM and MPI. Currently, on 512 processors of the Cray T3D the code runs at over 6 Gflops and on 256 processors of the IBM SP2 at 5 Gflops.

The existence of standards such as MPI and PVM on various platforms make porting of a program like QGMG written in message passing paradigm simple. The performance of these programs is also quite good since message passing encourages programming with data locality in mind which is essential for distributed memory machines. However, developing programs using these packages is onerous and it is difficult to ensure correctness of these programs since the message passing paradigm forces the programmer to be responsible for synchronization, data communication and data decomposition. For example, the QGMG code more than doubled in total number of lines when we converted the sequential code to a parallel one using PVM.

Alternatively one can write parallel programs using thread packages such as PTHREADS. Many parallelizing compilers like KSR FORTRAN are built on top of such thread packages. These packages support task parallelism using stateful threads and synchronization mechanisms such as barriers and semaphores. The user is responsible for determining the appropriate granularity, choosing the right synchronizations, and considering data locality. Because of the overhead cost of context switching and the data locality concerns, one normally creates only one thread per processor, each thread executing the same code over different data. A consequence of this SPMD style execution is that processors must idle if the thread running on it blocks due to communication latencies or due to synchronizations.

In this paper we present three implementations of the QGMG code: using MPI message passing, using a traditional thread package, and a new method using the DUDE runtime system. The goal of DUDE is to provide a user-level library or a compiler target that provides superior performance while freeing the user from the concerns of data locality, synchronization, scheduling and the appropriate granularity.

We show how this 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 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 [2] and distributed shared memory systems [3]. 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 [4]. Normally, dataflow execution models have been associated with dataflow processors [5, 6], but the macro-dataflow model has been implemented in software as well [7, 8]. Often, as in the case of MENTAT [9], 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-dataflow techniques with thread and loop-level scheduling systems.

The rest of the paper is organized as follows. We begin by describing the the QGMG application and implementations of it using MPI and the PTHREADS thread package. This will then motivate the description of our runtime system DUDE itself in §4. In section §5 we present some experimental results on the KSR-1. Finally we conclude in §6 with further extensions and future research directions.

2 The Quasi-Geostrophic Multi-Grid code

Planetary-scale fluid motions in the Earth's atmosphere and oceans are influenced by strong stable stratification and rapid planetary rotation. The appropriate equations of motion for this asymptotic regime are the Quasi-Geostrophic (QG) equations [10]. The extremely turbulent nature of planetary flows leads us to perform high-resolution numerical simulations of QG turbulence in an effort to better understand the large-scale flows which are so important to the Earth's climate. Due to stratification and rotation, QG flow is nearly incompressible in horizontal planes.

Over the last few years considerable effort has been invested into adapting and developing multigrid techniques for non-elliptic and singular perturbation problems, such as the flows found at high Reynolds number in stably stratified fluids. Integration of the QG equations requires solving an elliptic boundary-value problem in three dimensions even if the nonlinear advection equation for the potential vorticity is integrated explicitly. Moreover, the vertical-derivative term in this equation generally varies along the vertical coordinate. The QGMG code uses a multigrid algorithm, which is one of the best known methods for this problem. Furthermore, the nonlinear advection equation is discretized implicitly in time and the entire system solved simultaneously, employing the so-called Full Approximation Storage (FAS) version of the multigrid algorithm. Grid-coarsening is done only in the horizontal directions, with line relaxation in the vertical.

The Multigrid technique has received much attention due to it's importance and the challenge of parallelizing the algorithm [11] [12] [13] [14]. Multigrid algorithms are used to accelerate the convergence of relaxation methods like Gauss-Seidel for the numerical solution of partial differential equations. They achieve this by using a hierarchy of coarser grids with larger spacings to provide corrections to the approximate solution obtained on the finest grid. Thus there are three parts to any multigrid algorithm: relaxation (or smoothing) on a given grid, restriction from a fine to a coarser grid and interpolation (or prolonging) back from a coarse to a finer grid. The simplest restriction operator is to simply copy some of the points from the fine grid onto the coarse grid.

The algorithm we have described uses a V-cycle but there are many variants. Restricting and prolonging amounts to climbing up and down a pyramid of grids 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 1 shows the dependence relations between levels of a V-cycle for a one-dimensional problem. Each

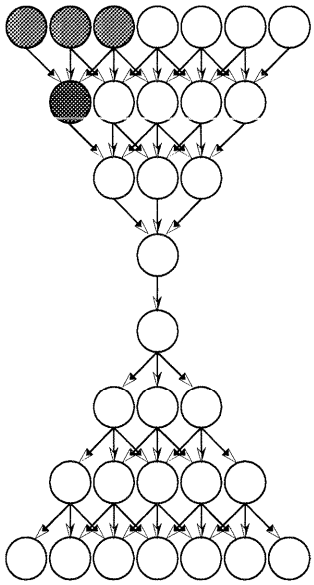


Figure 1: Dependence relations in a 1-dimensional multigrid application during a single V-cycle.

circle represents the execution of a single iteration of the relaxation function. Each level of the pyramid consists of three operations: smooth the red elements of the matrix, smooth the black 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. Normally, the dependence relations are satisfied by completing all iterations in each level before starting the next level. Figure 1 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.

The conventional parallel implementation of the multigrid method involves partitioning the finest grid matrix among the available processors. There is good processor utilization on fine grids but as the algorithm climbs up the pyramid to coarser grids, a majority of the processors need to idle at barrier synchronizations. In addition, a conventional runtime system does not support two multigrid solving tasks being run concurrently. 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 can allow processes to continue with the next operation immediately after their neighboring elements have been calculated, we will have better processor utilization. The proposed DUDE runtime system addresses these problems.

3 Threads

In the PTHREADS task parallel programming model, P threads, where P is the number of processors available, are created with each thread processing $\frac{N}{P}$ portion of the data. Figure 2 shows the flow of execution of parallel programs under the SPMD style programming using threads packages. At every barrier operation, all processes must synchronize before going on with the next operation. These are expensive synchronizations since the threads will arrive at different times at the barrier due to any one of the following reasons: page faults, communication latency due to remote memory address accesses, or execution of different portions of the code due to conditionals that are some function of the thread id. The completion time of the barrier synchronization is determined by the *slowest* thread. These costs do not seem to be warranted in light of the fact that most dependence relations in many applications do not involve all threads. In red-black relaxation for example, a point in the matrix is only dependent on its neighboring points, not the entire matrix. Figure 3 shows the possible flow of execution in many applications.

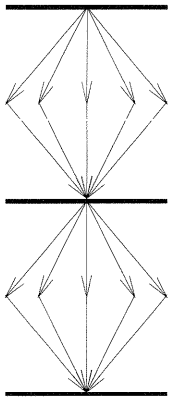


Figure 2: Traditional Fork-Join-Barrier structure.

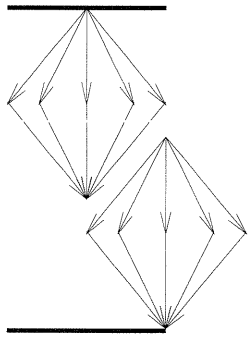


Figure 3: Inter-Loop execution.

4 DUDE

The DUDE runtime system is based on AWESIME [4] (A Widely Extensible Simulation Environment), an existing object-oriented runtime system for shared-address parallel architectures. The AWESIME library currently runs on workstations using DEC alpha AXP, SPARC, Intel 'x86', MIPS R3000 and Motorola 68K processors, as well as the Kendall Square Research 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 runtime system to implement the Definition-Use Description Environment (DUDE). 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 `Threads` 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, `Iterates` run to completion and are not context switched. `Iterates` cannot block on barriers or semaphores since they have no state, instead, explicit precedence information is used. Because `Iterates` 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 `Threads` 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 throughout 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 [15] 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 next in some detail.

4.1 Data Parallelism: Computation using Iterates

Computation using Iterates consists of 1) data decomposition, 2) data distribution, 3) data parallel operation, and 4) dependence satisfaction – all of which is handled by the runtime system. Data decomposition is the process of converting raw data such as a matrix into Iterate objects containing a method and a descriptor to a submatrix. A block, cyclic or any other decomposition method can be used. These Iterates are distributed among the available processors. The scheduling engine in each processor picks up these Iterate objects and does data parallel operations as specified in the method of the objects. When a Iterate object completes, the scheduler determines whether this completion of this object can enable other Iterates depending on it. If so, new Iterate objects are added to the queue. This process is repeated until the queue of objects is exhausted.

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 4 shows a sample code for the red-black relaxation that the application programmer or compiler provides. 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.

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. The scheduler calls the getContinuationDescs 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 the number of Iterates 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. If the constraints are satisfied, then the getContinuation method is used to instantiate the continuation and add this to the work heap. The runtime system performs all the synchronization required to ensure that the precedence constraints are satisfied. The application programmer or the compiler need only express the dependence information in the form of the getContinuationDescs 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 [16]. 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 [17], 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 subdivided into fixed sized chunks, with each chunk being represented by an Iterate object. If the processors experience load imbalance, as determined by a scheduling heuristic,

```
/*
================================================================
This describes the operation that the iterate computes on its
portion of data.
================================================================
*/
void RedRelax::main()
{
   for (short i = getSX(); i <= getEX(); i += getST()) {
     for (short j = getSY(); j <= getEY(); 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]);
     }
   }
}
/*
====================================================================
This method says that the continuation of the RedRelax iterate is the
BlackRelax iterate.
====================================================================
*/
BlackRelax *RedRelax::getContinuation(DESC desc)
{
  return new BlackRelax(desc.I,desc.J);
}
/*
============================================================================
This method gives the dependence info as a list of descriptors, one for
each arc leaving from this iterate to its children in the precedence graph.
============================================================================
*/
INDEX_DESC *RedRelax::getContinuationDescs()
{
    // get current index to this Iterate.
    int myI = getMyI();
    int myJ = getMyJ();
    INDEX_DESC *desc = FormIndexDesc({I,J}, {I-1,J}, {I+1,J}, {I,J-1}, {I,J+1});
    return desc;
}
```

Figure 4: Some methods from the red-black relaxation `Iterate`.

these fixed sized Iterates 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. Firstly, fixed size chunks simplify maintaining the dependence information, and make that process more efficient. Allowing variable sized chunks implies a less efficient data descriptor that takes a range of values instead of indices. We initially implemented such a structure, similar to the Data Access Descriptor [18], but found it was too slow in practice. Secondly, and more importantly, fixed size chunks allow the scheduler to establish an affinity between a chunk and the processor thereby 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 Iterates 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 Iterates to schedule is below a certain threshold, the CpuMux divides an Iterate, removing it from the queue only when all its 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 Iterates. 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.

Table 1: Speedups for 3d QGMG on KSR-1 and IBM SP2.

Processors	KSR-1	IBM SP2
1	1.0	1.0
2	1.4	1.8
4	2.8	3.1
8	1.9	4.4
16	1.8	10.1

5 Performance Results

We now give performance results for multigrid solvers running with MPI, with PTHREADS, and with DUDE. For the MPI results we have run the full 3d QGMG code for a problem size of 128x128x128 (timing only the multigrid part) to get the speedups shown in Figure 5. For PTHREADS and DUDE we have run the 2d multigrid kernel (of QGMG) solving a 1024x1024 matrix; these speedups are shown in Figure 6. We had hoped to have results for the entire QGMG code using the DUDE runtime system on the KSR-1 but unfortunately the machines we had access to have all stopped working, due to the untimely demise of the KSR corporation. We discuss each of the results we did get in turn.

5.1 MPI

MPI on the KSR-1 is/was relatively new and does not appear to be implemented very well as the speedups we obtained are somewhat disappointing – they are listed in the second column of Table 1. This should be contrasted with the IBM SP2 on which MPI is well implemented and the speedups for QGMG are much better – third column of Table 1. We expect that when/if MPI is tuned for the KSR-1 our speedups will improve, at least to those of the SP2. Therefore on the MPI performance graph, Figure 5, we have plotted both the KSR-1 and the SP2 speedups.

5.2 PTHREADS

The input matrix is divided into equal numbers of rows among the PTHREADS which are bound to

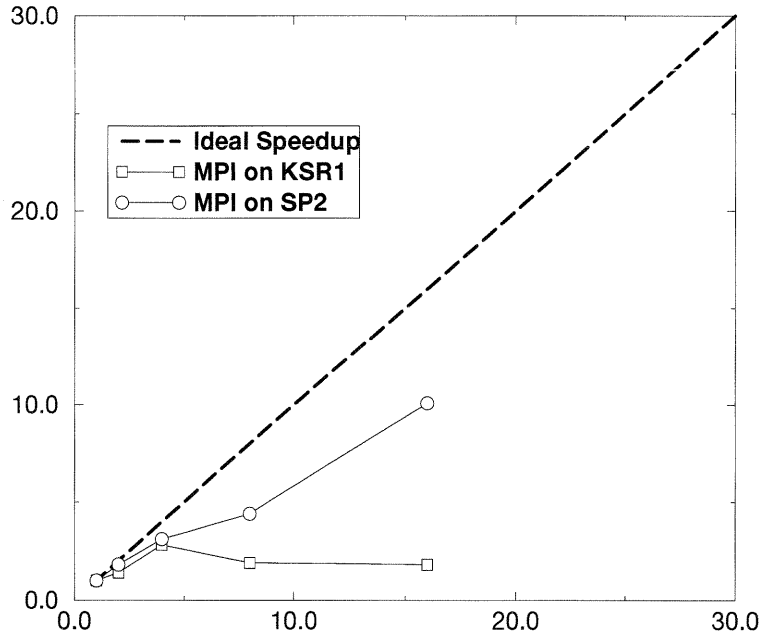


Figure 5: Speedups for full 3d QGMG code on the KSR-1 and IBM SP2.

physical processors. A barrier synchronization is used between each of the following operations: smooth red, smooth black, approximate, prolong and restrict. Due to the halving of matrix dimension at the next highest (coarsest) level, the number of processors participating is also halved to reduce the contention towards the top of the multigrid pyramid. Non-participating processors simply idle at the higher levels. The second performance graph, Figure 6, shows that the speedup on the KSR-1 is similar to what is obtained with MPI on the SP2 (Figure 5).

5.3 DUDE

To achieve both task and data parallelism, two threads (task parallelism) are created. Each thread starts a multigrid solver using `Iterates` (data parallelism). An `Iterate` class is created for each of the five operations: relax the red elements, relax the black elements, approximate, prolong and restrict. Initially only the RedRelax `Iterates` are created and added to the queues. As these complete and the precedence constraints are satisfied, BlackRelax `Iterates` (as specified in the `getContinuation` method of the RedRelax `Iterate` class) are enabled, and so on. After the Approximate `Iterate` completes, a choice of enabling either the Restrict `Iterate` or the Prolong `Iterate` must be made. This choice is made in the `getContinuation` method of the Approximate `Iterate`. Figure 6 shows that the multigrid solver using `Iterates` achieves superlinear speedup (due to locality) for small number of processors and near linear speedup for higher number of processors. Moreover the speedups with DUDE are significantly better than the speedups for both MPI and PTHREADS. Thus we expect that when the full 3d QGMG code is implemented with DUDE the speedups will be similarly impressive. It looks unlikely that we will be able to do this on the KSR-1 parallel computer, however as DUDE is portable we can do this on some other machine and in fact we are currently working on DEC alpha workstations.

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 prece-

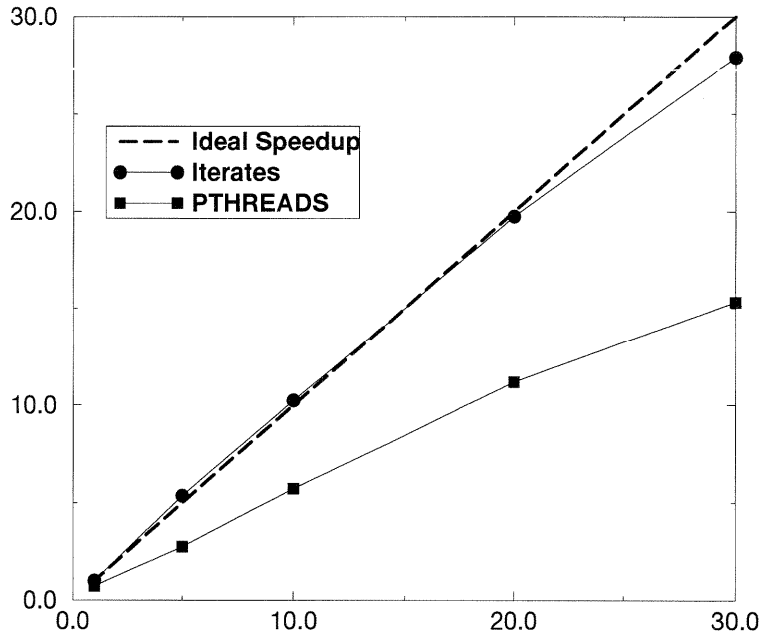


Figure 6: Speedups for 2d multigrid kernel on the KSR-1.

dence 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.

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 (for example, structures & fluids, oceans & atmosphere) concurrently. Combined thread and object parallelism is also important in applications such as adaptive mesh refinement, where data parallel operations are performed over a number of different arrays.

One feature not stressed 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, as well as allowing a great degree of extensibility.

Using information from profiling the application program, it is possible to determine its runtime behavior and choose 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 Threads 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.

We are also currently implementing the DUDE runtime system on DEC alpha workstations with distributed shared memory using DEC memory channels.

Acknowledgements

This work was funded in part by NSF Grand Challenge Applications Group Grant ASC-9217394, ARPA contract ARMY DABT63-94-C-0029 and an equipment grant from Digital Equipment Corporation.

References

[1] C.F. Baillie, J.C. McWilliams, J.B. Weiss and I. Yavneh. Implementation and Performance of a Grand Challenge 3d Quasi-Geostrophic Multi-Grid code on the Cray T3D and IBM SP2. In *Supercomputing '95 (to appear)*, 1995.

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

[3] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Winter 94 Usenix Conference*, pages 144–155, January 1994.

[4] D. 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.

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

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

[7] R. Babb. Parallel processing with large-grain data flow techniques. *IEEE Computer*, pages 55–61, July 1984.

[8] S. Ramaswamy and P. Banerjee. Processor allocation and scheduling of macro dataflow graphs on distributed memory multicomputers by the paradigm compiler. In *Proc. of the 1993 Intl. Conf. on Parallel Processing*, volume II-Software, pages II–134–II–138. CRC Press, August 1993.

[9] A.S. Grimshaw, E.A. West and W.R. Pearson. Easy to Use Object-Oriented Parallel Programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[10] I. Yavneh and J.C. McWilliams. Multigrid solution of stably stratified flows: the quasi-geostrophic equations. In *J. Sci. Comp. (to appear)*, 1995.

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

[12] D. Gannon and J. van Rosendale. On the structure of parallelism in a highly concurrent PDE solver. *J. Parallel Distributed Computing*, 3:106–135, 1986.

[13] P. Frederickson and O. McBryan. Parallel superconvergent multigrid. In *Proc. of the Third Copper Mountain Conf. on Multigird Methods*, pages 195–210. Marcel Dekker, 1989.

[14] S.N. Gupta, M. Zubair and C.E. Grosch. A Multigrid Algorithm for Parallel Computers: CPMG. *J. Sci. Comp.*, 7:263–279, 1992.

[15] D. Grunwald and S. Vajracharya. Efficient barriers for distributed shared memory computers. In *8th Intl. Parallel Processing Symposium*, pages 604–608. IEEE Computer Society, April 1994.

[16] 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.

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

[18] V. 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.