A Compiler Driven Execution Model for Irregular Applications

Arun Chauhan
Dept of Computer Science
Rice University
Houston, TX 77005
achauhan@cs.rice.edu

Kathleen Knobe
Cambridge Research Lab.
Compaq Computer Corp.
Cambridge, MA 02139
knobe@crl.dec.com

Abstract

Current parallelizing compiler technology does not handle irregular applications effectively. Approaches in the past have either focused on specific irregular applications to generate good code, or relied on fall-back generic techniques that are less efficient. This work proposes a compiler-driven approach that makes use of a novel run-time execution model for a cluster of computers. The model consists of data-flow style workers to enable load-balancing, and HPF-style data-blocking (or tiling) to divide computations into units of work. The run-time system is loosely based on Space-Time Memory abstraction developed at Compaq's Cambridge Research Lab, which enables blocks to be addressed totally independently of their physical location.

The model has several advantages. It allows a high degree of concurrency and high flexibility in scheduling computations. In some cases where a traditional compiler optimization (like loop fusion) can not be performed statically, this framework may lead to a natural dynamic equivalent. Use of workers enables good automatic load-balancing, while an advanced hierarchical design can achieve locality and computational scalability. We plan to leverage the dHPF compiler work at Rice University for a compiler front-end. Finally, the model is scalable in terms of space usage since space is required for only the currently relevant data blocks. The whole array may never exist at one time. Scalability in computation is being studied using hierarchical mapping of the model onto the underlying topology of a cluster.

We have developed this new approach via studying how it would work with two representative applications: blocked Cholesky factorization and Molecular Dynamics. In the paper we will detail the approach and provide evidence of its effectiveness on these two applications.

1. Introduction

Data parallel programming style has been very popular with the scientific community. It provides a programming paradigm that has become well understood. Several software tools developed over the years have made data-parallel style of programming easier and attractive. However, most of these tools – especially parallelizing compilers – have only been able to handle applications that fall in the class of *regular* applications. These applications exhibit regular patterns of computation and data access. This makes it possible for parallelizing compilers to reason about them, even in the presence of symbolic constants, and parallelize them effectively.

The applications that fall in *irregular* class exhibit data access patterns that are impossible to resolve at compile time. This usually gets manifested in source code in the form of index arrays that are used to access data arrays. Current compilers fail to handle such cases efficiently. Moreover, it is not easy to express data-parallelism in such applications.

This paper presents an execution model that aims to alleviate these problems. The motivation for the design of this model came from the experience of using Space-Time Memory [1] for an integrated task and data-parallel approach for multimedia applications [2, 3, 4]. With the increasing availability of high-speed low-latency networks, design choices that were earlier infeasible, may now become possible. Some such choices get reflected implicitly in the proposed model. Section 2 describes the execution model and its implementation. Section 3 discusses the use of the model for two representative applications. Section 4 describes the idea behind compiler integration of the execution model. Finally, the paper concludes with the discussion of current status and future work.

2. The Execution Model

Two properties characterize irregular applications:

- indirect data access patterns, and
- unpredictable computation requirements for a data sub-domain.

Any execution model that attempts to handle irregular applications must address these two issues.

We assume a specific class of applications. In particular, we assume the following:

- data-domain is amenable to block decomposition;
- computation can be partitioned based on the above block decomposition;
- there is sufficient block-level parallelism;
- data-dependencies between blocks can be determined either at compile-time or run-time;
- there is a simple mapping from data sub-domain to the iteration sub-space that generates that sub-domain.

While the above conditions might appear too restrictive, it turns out that several important scientific applications obey them.

The central idea is to decompose the data-domain into blocks and then decompose computation based on the blocked data decomposition. This results in a set of computational pieces of *work*, each dependent on a set of one or more data blocks or *data items*. At any time a global work-queue contains all pieces of work that are ready to be computed. A piece of work is ready to be computed when all the data-blocks required for its computation are available. This resolution is done by a component of the run-time system called *enabler*. With each piece of work are associated a set of input blocks and the computation to be performed. A set of identical workers, one for each physical processor, repeatedly obtain a piece of work from the queue, perform the associated computation and write the resulting data into a data *repository*. All data needed for a computation is read from this data repository.

2.1. High-Level Design

Figure 1 shows the overall structure of the execution model.

Data Repository The data repository contains all the data items that are currently *live*. Each data item is addressed by a global tuple that consists of array name and other attributes, as explained later in this section. The address is called *item-ID*. Inputs needed for a computation are read from the repository, and all the output generated as a result goes into the repository. A data-item is never replaced in the repository; new values generate new data-items with new item-IDs. This eliminates all but true data dependencies. As a data-item becomes "garbage" its space is reclaimed by the run-time system.

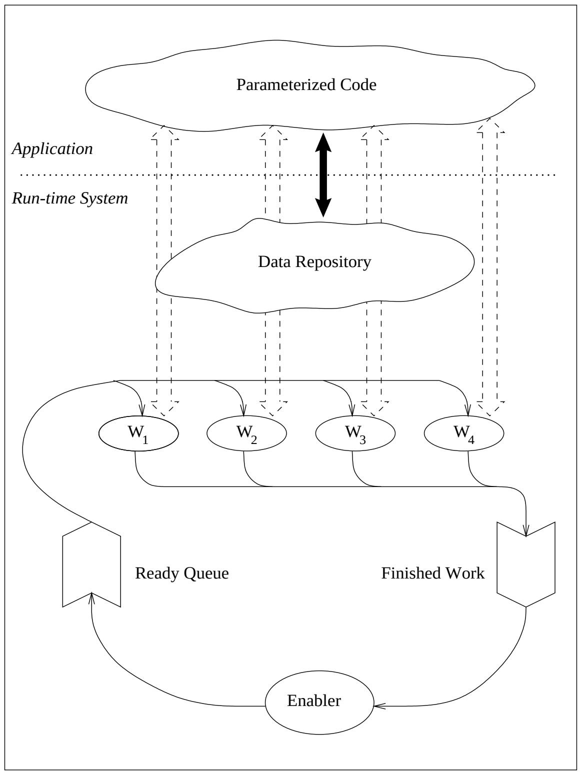

Figure 1. Logical structure of the Execution Model. W_i indicates the i^{th} worker.

Work Queue The work queue contains *work orders* that are ready to be executed. *Executing* a work-order means doing the computation indicated in it. A work-order is a complete specification of a computation including a label to identify a parameterized code block, and a list of all inputs in the form of data item-IDs. All the inputs needed for a work order in the work queue are guaranteed to be available in the data repository.

Workers All workers are identical workhorses that repeatedly obtain work orders from the work queue and execute them. Normally, there would be one worker for each available processor. Depending on the type of computation involved, as implied by the work-order, a worker invokes an appropriate piece of parameterized code in the application.

Finished Work Finished Work is a list of work-orders that have been executed. This serves as a mechanism to indicate the completion of a computation to the run-time system, and in particular, to the enabler.

Enabler The enabler is an active component of the run-time system. It is responsible for resolving data-dependencies at run-time. As soon as all the incoming data-items for a piece of work become available, the enabler generates a work-order and adds it to the work queue. An alternative way of looking at this function is that all the work-orders ever needed in the application exist in some virtual space and the enabler "enables" a work-order by moving it to the work queue when all the required input data for the work-order becomes available. In this sense, the enabler generates data-flow type computation pattern at the global level.

As described above, the model has serious potential bottlenecks in central work-queue. Locality is another important concern. Apart from the inherent gains in locality due to blocking, there needs to be a more sophisticated mechanism to exploit locality. These issues will be discussed in more details in section 8.

2.2. Implementation

As mentioned earlier, one inspiration for this model came from our experience with the Space-Time Memory (STM) system that was originally developed for multimedia applications. However, the current model is entirely different from STM. The primitives constituting the programming interface are radically different from those of STM. This results from the fact that scientific applications face issues that are different from the ones face by multimedia applications.

There are two major components of the proposed execution model:

1. compiler, and
2. run-time system.

Important components of the run-time system were described in 2.1 and figure 1. The run-time system has been implemented on top of the lower levels of Stampede [5] that are based on either a distributed object layer or a message massing layer.

Data Repository. Data repository is essentially an associative memory. It provides associative look-up based on data item-ID. Each item-ID can be an arbitrary tuple of scalar values, however, the size of the tuple is fixed for an application. This size is conveyed to the run-time once in the beginning. Notice that each data-item is a subsection of an array. Given our assumptions about the application, there must be a simple mapping from a given array sub-section to the iteration sub-space of the loop that generates the section. Moreover, since we restrict ourselves to rectangular sections, each data-item can be identified by the first iteration vector in the sequence that generates the data-item. Along with the array name and a loop-identifier, this iteration vector can be combined to form a unique data item-ID. For example, consider the code segment in figure 2. Assuming that the block size is 10x10, data item that corresponds to array section A(1:10,11:21) generated in loop 2, could be identified with a tuple of the form <array name, loop label, {loop-vector}>. For this data-item the item-ID would then be <'A', 2, {1,11}>.

With each data item are associated two attributes: the item ID and a *garbage ref-count*. Item ID, described above, consists of a tuple that identifies the data item globally. The garbage ref-count indicates the number of times a data-item will be read before it becomes garbage. This number is specified by the application whenever a new data-item is created and written into the data repository. Subsequently, every time the data-item is read, its garbage ref-count is decremented. As soon as the count reaches zero, its space can be reclaimed. This is possible because a data-item is never modified. An important result of this mechanism is that arrays never reside at any central place and only the "live" data ever exist in the system. Moreover, the data-items are distributed across processors, thus making the design highly scalable with respect to space requirements.

Work Queue. Logically, there is a single shared queue from which all workers read work-orders. This greatly simplifies the logic that each worker must implement and also provides opportunities to tune the actual implementation according to the underlying architecture. Currently, the work-queue is implemented as a physically single work-queue. To be scalable, it needs to be hierarchical in structure that

```
loop1:   do i = 1, 1000
             do j = 1, 1000
                 C(i, j) = C(i, j) * D(i, j)
             enddo
         enddo

loop 2:  do i = 1, 1000
             do j = 1, 1000
                 A(i, j) = B(i, j) + C(i, j)
             enddo
         enddo
```

Figure 2. Example code to illustrate the Execution Model.

matches with the underlying topology of the network. However, any such implementation details are hidden from the top level model. The idea of hierarchical queues is discussed in more details in section 8.

Workers. Usually, there would be one worker per processor. Each worker is a simple driver that obtains a work-order from the work-queue, reads all the input data needed to execute that work-order and then invokes the appropriate code segment to perform the associated computation. This provides a load-balanced execution model. The code that gets invoked from within a worker is generated by the compiler. It is nothing but a parameterized version of a code segment in the original source that has been transformed to work on blocks of data. Currently this is done by hand. However, the worker itself is completely independent of the specific application and is a part of the pre-compiled run-time system.

Finished Work. Finished Work is the list of work-orders that have been executed. The enabler uses this information to resolve dependencies and add new work-orders to the work-queue. Dependencies are conveyed to the run-time system "on the fly" through the `insert_WO` call, as would become clearer in 2.3.

Enabler. Analogous to the garbage ref-count for a data-item, an *enabling ref-count* is associated with each work-order. It refers to the number of inputs a work-order needs for it to be executed. When a data-item is computed, it triggers one or more work-orders that would use that data-item. This information is conveyed to the enabler for each work-order thus triggered. The enabler determines if all the inputs for that work-order are ready; if so, it places the work-order in the ready queue.

2.3. Programming Interface

Compiler-transformed application code interacts with the run-time system through a set of library calls. The calls can be divided into three broad categories:

registration calls These are a set of functions that are called once in the beginning of the program. They are used to convey application specific information to the run-time system. This includes size of the item-ID tuple and pointers to various code segments used to execute work-orders.

execution related calls The central part of the Application Programming Interface (API) is the set of functions that read and write data from the repository. Workers implicitly call parameterized code segments. These code segments use the data repository related API calls to exchange data with the repository. Finally, another API call enables the code segments to convey to the run-time system the completion of a work-order as well as further work-orders that directly depend on the data produced.

support functions There is a set of support functions that serve primarily to "construct" and "de-construct" work-orders, item-IDs, etc.

`register_dItem_t`	
function	convey the tuple size and other attributes that constitute the item-ID to the run-time system
arguments	The function takes the number of fields in the tuple and maximum value for each field of the tuple. Each tuple field is restricted to scalar values – the maximum value is used to compress each field into minimum number of bits.
`register_executor`	
function	convey the code segments (represented by function pointers) to the run-time system
arguments	The sole argument is a pointer to a function.
`read_data`	
function	read data from repository; each read decrements the garbage ref-count associated with the data-item
arguments	Work-order and pointers to buffers to read all the required data for the work-order.
`write_data`	
function	write data to repository
arguments	Pointer to the buffer containing data, the item-ID for the data-item, and the garbage ref-count. The ref-count allows the run-time system to reclaim space when the data-item is no longer live.
`insert_WO`	
function	generate a new work-order (that may not be ready to be executed yet)
arguments	Item-ID of a newly generated data-item and the total enabling count for the work-order. The enabling count (analogous to the garbage ref-count) enables the run-time system to decide when all the inputs become ready for a work-order and move it to the ready queue.

Table 1. Major API functions along with their important arguments.

Table 1 lists the main API functions along with a brief description of their arguments and their functionality. One important observation is that even though the API is targetted at compiler generated codes, it is simple to use. This has two advantages: it enables testing applications without any compiler support and it keeps the code-generation phase in the compiler simple. A somewhat more complex requirement of the compiler is transforming the source code segments into data-flow type parameterized functions.

Most API functions are self-explanatory, however, `insert_WO` merits some elaboration. The dependence information for a work-order gets conveyed to the run-time system through `insert_WO` call. The function is called whenever a new data-item is generated that would be used in a subsequent work-order. It is called once for each such work-order. Each call also conveys one of the inputs to the specified work-order, i.e., the data-item just generated. The enabler maintains information about partly ready work-orders. Once all inputs for a work-order have been conveyed to it through `insert_WO` calls, the work-order is moved to the ready queue.

3. Applying the Model

3.1. Cholesky Factorization

Cholesky factorization is not completely irregular, but does exhibit complicated communication patterns as well as characteristics that make it difficult to load-balance. This provides a useful test-bed for the Execution Model.

The problem of Cholesky factorization is to decompose a given symmetric positive definite matrix, M, into a lower triangular matrix, L, so that $M = L.L^T$. Figure 3 shows the basic, right-looking, Cholesky factorization algorithm as well as the blocked version of the same algorithm; the blocked version reduces to the basic one for a block size of unity. This is a well known blocked-Cholesky algorithm. Even though the computation and the data reference patterns are completely predictable, it is hard to come up with a static data distribution that would lead to good load-balance and keep communication patterns simple. This was the motivation behind using our Execution Model to implement Cholesky factorization.

Figure 4 shows the computation pattern of two initial iterations. Notice that in each iteration there are three distinct types of operations:

1. simple Cholesky factorization of the topmost triangular block in the left column,
2. multiplication operation for all the remaining blocks in the leftmost column, and
3. "update" operation for all the blocks in the remaining triangular matrix.

The algorithm then proceeds to the next iteration by repeating similar three-step computations on the smaller triangular matrix that results by ignoring the leftmost column of blocks.

```
subroutine simple_Cholesky (M)
// Assuming the matrix, M, is NxN.
// Cholesky is computed in place.
// Only the lower triangular part is used.
do k = 1, N
    M(k,k) = sqrt(M(k,k))
    do i = k+1, N
        M(i,k) = M(i,k) / M(k,k)
    enddo
    do j = k+1, N
        do i = j, N
            M(i,j) = M(i,j) - M(k,i)*M(k,j)
        enddo
    enddo
enddo

subroutine blocked_Cholesky (M)
// assuming the matrix, M, is TxT blocks.
// Cholesky is computed in place.
// Only the lower triangular part is used.
// M(i,j) refers to block (i,j).
// Arithmetic operators are matrix operations.
do k = 1, T
    call simple_Cholesky(M(k,k))
    do i = k+1, T
        M(i,k) = M(i,k) * inv(M(k,k))
    enddo
    do j = k+1, T
        do i = j, T
            M(i,j) = M(i,j) - M(k,i)*M(k,j)
        enddo
    enddo
enddo
```

Figure 3. Cholesky factorization.

Similarly shaded areas in the figure indicate similar and *independent* computations. Thick arrows between the shaded areas show data dependencies between these regions. Parallelism exists at several levels:

1. all blocks shaded similarly can be processed in parallel;
2. many blocks in the shaded area corresponding to the next step become ready to compute even before the previous area has been completely computed; the computation for those blocks can be performed concurrently;

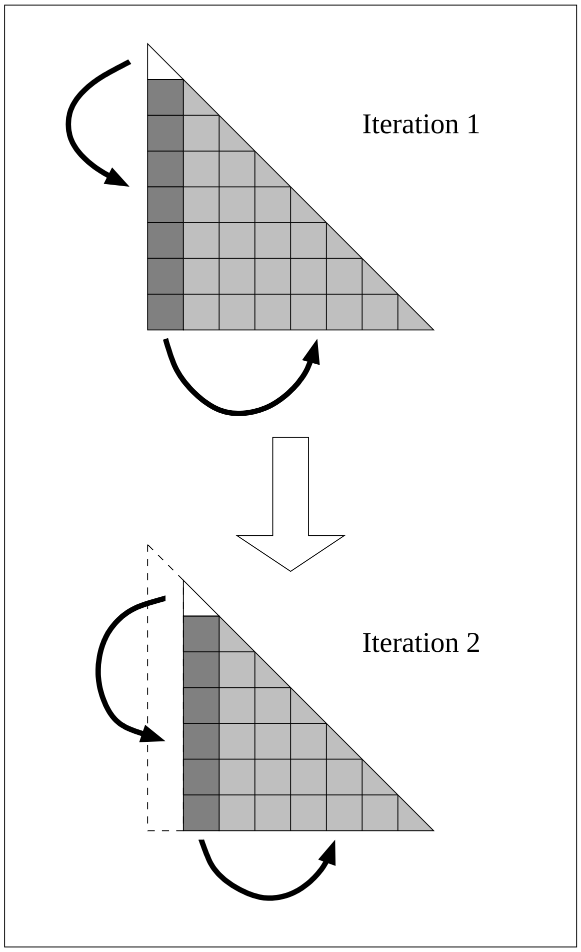

Figure 4. Computation pattern in the first two iterations of blocked Cholesky factorization.

3. computation for the next iteration can be started even before the previous iteration has finished.

Our execution model is capable of exposing, and using, all three levels of parallelisms. The simplest way to map this application onto the execution model is to consider each block of matrix as a data-item. Further, recall that new values for a matrix block give rise to new data-items in our model – a data-item is never modified. The work-orders naturally correspond to the computations for data-items. There are three types of work-orders for the three types of computations.

Figure 5 illustrates the transformed Cholesky factorization code to use the proposed execution model. For the sake of clarity, the figure omits several details and presents the transformations for only the initialization code and one type of computation in pseudo-code form.

```
subroutine main_Cholesky (M)
    register item_type tuple size as 3 // 3 is the max loop nest depth
    register type_1_computation
    register type_2_computation
    register type_3_computation
end subroutine

subroutine type_1_computation (work_order W)
    Matrix A
    call read_data(A, W)
    call simple_Cholesky (A)
    let k = row and column of the block A
    let n = number of blocks along a dimension in the original matrix
    garbage_ref_count = n - k // A is used by (n-k) type-2 computations
    call write_data(A, garbage_ref_count)
    do i = k+1, n
        let ID_computed = item ID for block A
        let ID_2compute = item ID for block (i,k)
        enable_count = 2 // two inputs needed for type 2 computation
        call insert_WO (enable_count, ID_computed, ID_2compute)
    enddo
end subroutine
```

Figure 5. Transformed Cholesky factorization to use the proposed execution model.

3.2. Molecular Dynamics

Molecular Dynamics is an application that simulates a system of a large number of molecules evolving in time under the influence of mutual electro-static forces. The algorithm starts out by initializing particle velocities and forces and building a list of interactions consisting of pairs of “neighboring” particles as defined by a threshold radius. Rest of the algorithm iterates inside a time-step loop. First, the algorithm updates particle coordinates. It then iterates over all pairs of interactions and updates the mutual electro-static forces. Next, the new particle velocities are computed based on updated forces. Finally, the total kinetic energy and average velocity of the system are computed through a global sum operation. This completes one iteration of the algorithm. Clearly, the complexity of each iteration is bounded by the number of interaction pairs and the number of particles. Every fixed number of iterations the algorithm recomputes the list of neighbors. This is a $O(n^2)$ operation, where n is the number of particles. The high-level pseudo-code is shown in figure 6.

This non-hierarchical N-body simulation code provides a good example of highly irregular application. It is hard to parallelize using conventional mechanisms. To parallelize this application using our execution model, we make the following observations.

- There are three types of computations involved: force

```
program moldyn
    build neighbor list
    initialize particle coordinates, velocities, and forces
    for t = 1, MAX_TIMESTEP do
        if (mod(t,20) .eq. 0) then
            re-build neighbor list
        endif
        for each pair of neighbors do
            update forces
        enddo
        for each particle do
            update velocity
        enddo
        compute system's kinetic energy
        compute average velocity
    enddo
end program
```

Figure 6. Pseudo-code for Molecular Dynamics.

computation that is per interaction pair; velocity computation that is per particle; and global sums (kinetic energy and average velocity) that translate to reduction operations.

- Updating the neighbor list involves all-to-all communication of particle coordinates.
- All data is associated with particles and none is associated with an interaction pair.

Based on the above observations one possible strategy is as follows. Each force computation or velocity computation is treated as a work-order. Each particle is treated as a data-item with the particle attributes (velocity, coordinates, and force) being the data. Computation of forces, velocities, and coordinates can, thus, be easily performed within the worker driven execution model described earlier. To compute the global sum for computing kinetic energy and velocity average, work-orders are created to compute partial sums. Each worker computes the partial sum using as many values as it can. Finally, a single work-order is created that has as many inputs as the number of processors, to compute the final value. Notice that this final sum can be carried out by any one processor while the others can proceed to start computations for the next iteration without waiting. This strategy allows concurrency to be exploited across computational phases within a single time-step as well as across time-steps. This effect is similar to what the model achieved for Cholesky factorization.

The only step that now remains to be parallelized is the re-computation of neighbors. Unfortunately, this is the bottleneck in this algorithm and it remains the bottleneck in parallelization too. This step involves large amounts of communications. One way to ameliorate the situation is to exploit locality. The other is to use a better algorithm.

3.3. Hierarchical N-body Simulation

Hierarchical N-body algorithm is a big improvement over the straightforward algorithm used in Molecular Dynamics. The best known adaptive methods achieve $O(n)$ time complexity, where n is the number of bodies. We are in the process of evaluating an adaptive N-body code using our execution model.

4. Compiler

HPF is among the most popular of the several data-parallel programming models. It has been highly studied and there is a good compiler support available for HPF at research as well as commercial level. One big advantage of HPF paradigm is that it enables the application writer to think and program in the well understood sequential way. Secondly, a number of applications have already been ported to HPF, which provide useful comparative benchmarks. Finally, the compiler development effort at Rice University has resulted in an extensive infrastructure for HPF compilation that we intend to utilize to test our ideas. For these reasons, even though the proposed execution model would work with most parallelizing compilers and data-parallel environments, HPF is our preferred environment.

The proposed model involves re-writing the source code and inserting calls to the run-time API with appropriate parameters. This is a tedious process to do by hand, and therefore, also prone to errors. Current compiler technology is capable of handling these transformations. The missing link is to characterize applications that would benefit from this model and encode it as a compiler algorithm. This would enable a parallelizing compiler to transform the source for using this run-time system only when it is profitable to do so.

5. Current Status

Current implementation of the run-time system is on top of the lower layers of Stampede. This uses Stampede's *channel* mechanism to implement data repository, and a separate Stampede thread to implement the "enabler". Each worker is a Stampede thread, one per physical processor. Although, for the sake of efficiency, all threads on a single

SMP run in the same address space, the run-time system is independent of this fact.

Cholesky factorization has been written and tested on this system. Molecular Dynamics is at an advanced stage of porting. We hope to evaluate these two applications and have some performance results ready by the time of the workshop.

From the past experience with Stampede, and with some preliminary performance measurements, our estimate is that certain Stampede related primitives used in implementing the run-time system are too inefficient. There are several characteristics of our execution model that allow a much more efficient implementation of these primitives.

- While Stampede supports a sophisticated garbage collection mechanism that is scheduled periodically in the background, this is not needed for our purposes. All the data-items have well known lives that are completely determined by the garbage reference counts. This would enable several optimizations by getting rid of certain overheads related to garbage collection.

- Many underlying data structures being used by Stampede are tuned for streaming multimedia applications. In particular, they are well suited for sequential access. In scientific applications, the accesses are fairly random. We believe that re-tooling these underlying data structures can lead to significant performance gains.

- The current implementation of enabler mechanism resides completely above the Stampede layer. Since, it is really a lower level functionality, by pushing it down, some improvements in performance are possible.

- Finally, the entire Stampede layer, that is itself built on top of an underlying message passing or distributed object layer, introduces unnecessary overheads. One immediate goal after the rapid prototyping stage is to bypass this layer completely and implement the run-time system directly over the underlying layers.

Apart from these immediate measures, the future work section discusses other ideas that can lead to far reaching performance benefits in the long run and that can help strengthen the execution model.

6. Related Work

One of the pioneering works to handle irregular applications automatically was done as the CHAOS project at University of Maryland [6]. The system consisted of a run-time library that provided primitives to resolve data locations at run-time. Data could be "re-mapped" at run-time to improve load-balance. Inspector-executor strategy was utilized to compute communication requirements for a loop and arrive at an optimized communication schedule.

The inspector-executor strategy has inherent overhead in that a computation loop must be executed, without the computation part, as a pre-processing step. In addition, a two-step process is necessary to compute the location of a piece of memory. As a result, load-balancing must be traded off against data re-distribution and re-computing communication schedules.

Some work on encoding irregular applications in HPF was done by Charlie Hu et al.[7]. They succeeded in encoding adaptive hierarchical N-body problem in HPF by using space-filling curves . However, they did not get very good performance with available compilers. A later study by Collin McCurdy and John Mellor-Crummey[8] found that to achieve acceptable performance with HPF encoded N-body code it was necessary to use *extrinsic* procedures – a mechanism to slip into lower levels of programming that destroys the abstract HPF programming paradigm.

The SMARTS project at Los Alamos National Laboratory [9] is similar in that it attempts to achieve higher parallelization by scheduling iteration sub-spaces or "iterates" independently. This has the flavor of data-flow type computations, but they do not address irregular applications. Moreover, their work is restricted to a single SMP, so the data addressing issue has not yet been handled.

The work that does address the data addressing issue is the LINDA project at Yale University [10]. LINDA has a very general tuple-based resource addressing mechanism. However, unlike the model presented in this paper, LINDA's API is intended to be directly used by the application writer and it aims to provide a general parallel-programming model. As a result there are issues related to protection, aliases, etc that do not arise in our case. In addition, the primitives supported by LINDA are quite different reflecting the difference in desired goals. For example, it provides a mechanism to look up tuples based on wildcards; this is a generality not required by our system, thus simplifying its implementation.

7. Conclusion

We have presented an execution model for irregular applications. The model works by combining HPF-style data parallel compiler with a run-time system support to provide a software layer that enables automatic handling of irregular applications that have been very hard to handle so far. Two key elements of the model are location independent tuple based mechanism to address data, and data-flow style worker based run-time system for automatic load-balancing. The model works seamlessly both within a SMP and across a cluster of them.

We have evaluated our model in the context of two applications: Cholesky factorization and Molecular Dynamics. At present we address a restricted class of irregular applications whose computation can be blocked and that exhibit sufficient parallelism at the block level. Some important scientific applications fall in this category.

Our immediate future goals are to do extensive performance evaluations on these and other applications, and address the issues of locality, automatic code generation, and scalability.

8. Future Work

There are two important aspects of the execution model that need to be addressed in future:

- Hierarchical model to enhance data-locality and make the model scalable.
- Compiler algorithms to integrate the system with HPF compiler.

Performance of the run-time system is the key in making the system practical. One important consideration is data locality. The issue of locality arises at two levels: availability of data on local processor as against obtaining them remotely; making maximum use of the local memory cache.

To address the first, one possible approach is to design a hierarchical system that takes into account the underlying network topology. The hierarchical system would implement a cache system for data-repository. The cache would store local copies of the data that is read. As all the data is read-only, cache coherency would not be an issue. However, keeping the garbage reference counts consistent is an issue that needs to be solved.

To make optimal use of cache, it is desirable to make as much use as possible of the data once read from memory, before it is discarded. This points to some strategy to order work-orders is a way that reuses data that is already in cache. Such a strategy would use multi-level work queues.

Not only would a hierarchical design address the issues of scalability and locality, it would also allow the model to be used on a heterogeneous cluster of machines. These enhancements would leave the API unchanged and hence still provide a simple view to the application. This enables experimenting with and tuning the run-time system without requiring any change at the compiler-end.

At the compiler-end, we need an algorithm for evaluating and transforming applications to use the execution model. The idea is that the user would program in HPF as usual, while the compiler evaluates the applicability of the run-time system for the application. Admittedly, this is an ambitious goal; as an intermediate step, we may have to relax our goals and seek the help of special compiler directives for semi-automatic handling.

9. Acknowledgments

We thank Collin McCurdy and John Mellor-Crummey for making their code available; thanks to Charlie Hu for the HPF version of hierarchical N-body code. Shamik Sharma, from University of Maryland, wrote the original Molecular Dynamics code. Finally, we would like to thank Compaq Computer Corp.'s Cambridge Research Lab for making their facilities available.

References

[1] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe, "Space-Time Memory: A parallel programming abstraction for interactive multimedia applications," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, May 1999.

[2] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran, "Dynamic task and data parallelism using Space-Time Memory," Tech. Rep. 98/10, Compaq Computer Corp., Cambridge Research Lab, Nov. 1998.

[3] K. Knobe, J. M. Rehg, R. S. Nikhil, U. Ramachandran, and A. Chauhan, "Integrated task and data parallel support for dynamic applications," *Scientific Programming*, vol. 7, pp. 289–302, 1999.

[4] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran, "Scheduling constrained dynamic applications on clusters," in *Proceedings of the ACM / IEEE SC Conference on High Performance Networking and Computing*, Nov. 1999.

[5] R. S. Nikhil, U. Ramachandran, J. M. Rehg, J. R. H. Halstead, C. F. Joerg, and L. Kontothanassis, "Stampede: A programming system for emerging scalable interactive multimedia applications," in *Eleventh International Workshop on Languages and Compilers for Parallel Computing*, Aug. 1998.

[6] S. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptive irregular problems," in *Proceedings of the ACM / IEEE SC Conference on High Performance Networking and Computing*, Nov. 1994.

[7] Y. C. Hu, S. L. Johnsson, and S.-H. Teng, "High performance Fortran for highly irregular problems," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, June 1997.

[8] C. McCurdy and J. Mellor-Crummey, “An evaluation of computing paradigms for N-body simulations on distributed memory architectures,” in *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, May 1999.

[9] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shendey, R. Oldehoeft, and S. Smith, “SMARTS: Exploiting temporal locality and parallelism through vertical execution,” in *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, 1999.

[10] N. J. Carriero, Jr., *Implementation of Tuple Space Machines*. PhD thesis, Yale University, Dec. 1987. YALEU/DCS/RR-567.