International Conference on Parallel Processing 1993.

LOCALITY AND LOOP SCHEDULING ON NUMA MULTIPROCESSORS

Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik
Computer Systems Research Institute
University of Toronto
Toronto ON M5S 1A4
CANADA

Abstract

An important issue in the parallel execution of loops is how to partition and schedule the loops onto the available processors. While most existing dynamic scheduling algorithms manage load imbalances well, they fail to take locality into account and therefore perform poorly on parallel systems with non-uniform memory access times. In this paper, we propose a new loop scheduling algorithm, Locality-based Dynamic Scheduling (LDS), that exploits locality, and dynamically balances the load.
Key Words: Locality, Loop Scheduling, NUMA Multiprocessors, Data Partitioning, Locality-based Dynamic Scheduling.

1 Introduction

Loops are a major source of parallelism for todays parallelizing compilers. An important issue in the parallel execution of loops is how to partition and schedule the loops onto the available processors. A number of algorithms have been proposed for this purpose. For example, *static scheduling* algorithms such as block, cyclic, and block-cyclic scheduling, partition the loop into fixed-sized chunks and distribute the chunks evenly across processors statically at the beginning of the computation. *Dynamic scheduling* algorithms, on the other hand, assign the loop partitions at run time, depending on the speed and progress of the processors. For example, self scheduling [5] partitions the loop into fixed size chunks, which are conceptually organized in a single system-wide queue, and each processor obtains a new chunk from the queue when it has completed its previous chunk. More recent proposals, such as guided self scheduling (GSS) [10], factoring [4], and trapezoid [13], vary the size of the chunks; they start with large chunks in order to reduce the overhead in accessing the central queue and then progressively use smaller chunks in order to maintain good load balance. These scheduling algorithms are described in detail in Section 2.

All of the loop scheduling algorithms listed above assume a shared memory architecture with uniform memory access (UMA) costs, and hence need not take data locality into consideration. However, many of the more modern, especially scalable, shared memory multiprocessors have non-uniform memory access (NUMA) cost; *i.e.*, the cost of accessing memory increases with the distance between the accessing processor and the target memory. Examples of multiprocessors with non-uniform memory access costs include DASH [6], Hector [14], BBN [12], RP3 [7], and Cedar [8]. In these systems, data locality is important for good application performance, and the loop scheduling algorithms should take this into account.

In this paper, we introduce a new loop scheduling algorithm that takes data locality into consideration, and compare its performance with other well known scheduling algorithms. The next section describes some of the existing loop scheduling algorithms. Section 3 describes data locality and why it is an important factor that cannot be neglected in loop scheduling algorithms. In particular, we argue that data locality is important even in systems with hardware-based cache coherence and in cache-only-memory architectures (COMA), such as the KSR [1]. The locality based dynamic scheduling (LDS) algorithm we propose in this paper is described in Section 4, and compared against the affinity scheduling algorithm developed at the University of Rochester, the only other loop scheduling algorithm we are aware of that also takes memory access locality into consideration. In Section 5, the results of experiments comparing LDS to the other scheduling algorithms are described.

2 Scheduling Algorithms

2.1 Static Scheduling

Static scheduling algorithms, such as block scheduling, cyclic scheduling, and block cyclic scheduling, assign a fixed number of loop iterations to each processor.

Block scheduling divides the loop into blocks of $\lceil N/P \rceil$ iterations, where N is the number of iterations and P is the number of processors. Each processor is assigned a separate block. If the amount of computation performed by each iteration differs, then block scheduling can perform poorly because of load imbalance. For example, in an iteration space where the amount of computation per iteration increases linearly, the first few blocks will entail very little computation while the latter ones will involve much more. The first few processors will therefore finish their computations early and have to wait for others to complete, resulting in poor speedup.

Cyclic scheduling assigns loop iterations to processors in a cyclic order, so that processor p will execute the iterations $p, p+P, p+2P, \ldots$, where P is again the number of processors executing the loop. In contrast to block scheduling, cyclic scheduling obtains better load balance for triangular iteration spaces and other iteration spaces where the amount of computation increases/decreases linearly with the iterations.

Block cyclic scheduling is a compromise between block scheduling and cyclic scheduling. This algorithm assigns blocks of a fixed size to processors in a round robin fashion. If the block size is equal to one, then block-cyclic scheduling degenerates to cyclic scheduling and if the block size is $\lceil N/P \rceil$, then block-cyclic scheduling is same as block scheduling. Hence, block cyclic scheduling forms a continuum between block and cyclic scheduling algorithms.

The static algorithms ignore the fact that the amount of computation performed per iteration may differ, or that it cannot always be determined a priori (for example, the amount of computation could be dependent on the data). Moreover, the speed of each processor may also differ because of multitasking interference. Therefore static scheduling often suffers from load imbalance, resulting in poor speedup.

2.2 Dynamic Scheduling

If the imbalance becomes large, then it is necessary to dynamically adjust the work assigned to each processor at run-time in order to balance the load. This is done by grouping together one or more iterations to form subtasks which are dynamically allocated and executed by the processors. The subtasks need not be of fixed granularity, and in fact the granularity could vary dynamically. If the granularity is very large (N/P iterations per subtask) then we effectively have block scheduling. On the other hand, if the granularity is very small, then the data structure controlling the subtasks could become a bottleneck because of the number of accesses each processor must perform to this structure. Self scheduling, guided self scheduling, factoring, and the trapezoid method are examples of dynamic scheduling algorithms.

3 Locality and Scheduling

Self scheduling partitions the loops into subtasks containing one or more iterations [5]. Each processor then continuously allocates and executes one subtask at a time until no subtasks are left for processing. If the number of iterations per subtask is fixed and greater than one, then this scheduling strategy is generally referred to as *fixed-size chunking* [5].

With fixed-size chunking it can be difficult to choose the correct granularity. Small granularity increases the overhead of accessing the data structure controlling the subtasks that still need to be executed. Larger granularity can lead to load imbalances when the last set of subtasks is being executed. Hence in the algorithms that follow, the size of the subtasks is dynamically adjusted with the progress of the computation.

Guided self scheduling(GSS) uses a subtask granularity of $\lceil n/P \rceil$ iterations, where n is the total number of remaining iterations [10]. With this algorithm, the subtasks are composed of a large number of iterations at the start of the computation, then progressively fewer until the size is one. In this scheme, there will be at least $P-1$ subtasks consisting of only one iteration, and each will be executed independently. When the execution time of the iterations differ, it is possible that an early subtask could be so large that it does not complete by the time all other subtasks have completed [4]; this load imbalance problem is addressed by the factoring algorithm.

Factoring is similar to GSS in that the size of the subtask decreases as the computation progresses, but it assigns $\lceil n/(2P) \rceil$ iterations to P consecutive subtasks, where n is equal to the number of remaining iterations at the beginning of these allocations [4]. Hence P consecutive subtasks will be of the same size, before the granularity is decreased. If the variance of the amount of computation performed by each iter-

Scheme	No. of Iterations = 500 and P = 4
GSS	125 94 71 53 40 30 22 17 12 9 7 5 4 3 2 2 1 1 1 1
Factoring	63 63 63 63 31 31 31 31 16 16 16 16 8 8 8 8 4 4 4 4 2 2 2 2 1 1 1 1
Trapezoid	62 58 54 50 46 42 38 34 30 26 22 18 14 8
LDS	63 55 48 42 37 32 28 25 22 19 17 14 13 11 10 8 7 7 6 5 4 4 3 3 3 2 2 2 1 1 1 1 1 1 1 1

Table 1: Subtask Sizes for dynamic scheduling algorithms. The GSS, factoring and trapezoid algorithms are described in Section 2. LDS is described in Section 4.

Arch.	Cache	Local Mem.	Remote Mem.
Hector	1	10	24
DASH	1	22	61
RP3	1	10	15

Table 2: Latency for memory read operation in processor clocks

ation is large, then factoring performs better than GSS [4].

The **Trapezoid** method also assigns a decreasing number of iterations to subtasks and thus is a variation of GSS. In this case, however, the subtask size decreases linearly instead of exponentially [13]. The total number of iterations, N, is partitioned into $S = \lceil 2N/(f+1) \rceil$ subtasks, where $f = \lfloor N/(2P) \rfloor$ is the size of the first task. Consecutive subtasks differ by $\lfloor (f-1)/(S-1) \rfloor$ iterations.

For comparison, the subtask sizes employed by the dynamic scheduling algorithms GSS, factoring, and the trapezoid method for a problem size with 500 iterations executing on four processors is given in Table 1.

In NUMA systems, managing data locality is important due to the increased cost of accessing remote memory. Table 2 shows the difference between remote memory access costs and local memory access costs for different architectures configured with 64 processors. On these systems having most of the accessed data local to the accessing processor can be a major factor in improving performance [2, 3, 11].

In parallelizing a loop, it is important to consider the partitioning of both the data space and the loop iteration space, and how both are mapped onto the processors. For good performance, it is essential that the loop partitions and scheduling match the data partitions. Best performance is achieved when all data required by a loop partition is local to the processor on which the partition is scheduled. A mismatch in the scheduling of the loop partitions and the data partitions can have a heavy performance penalty on NUMA multiprocessors, as will be shown in Section 5.

Consider for example the simple loop shown in Figure 1. The iteration space and the data space are two dimensional. Because the inner loop j is sequential, the data space must also be partitioned row-wise and implicitly the iteration space must also partitioned row-wise. Because of the simple reference pattern, the loop and data partitions match. If the loop partition $i = 0$ is scheduled on the processor which has the row $A[0][*]$, then all the accesses to A are local. Otherwise all the accesses would be non-local. We call the static scheduling algorithm *Block-D* if both the data partitioning and the loop scheduling occur in blocks. (Analogously we use the terms *Cyclic-D* and *Block-cyclic-D* if both data partitioning and the loop scheduling match.)

```
parallel_for(i=0; i < N; i++)
       for(j=0; j< N; j++)
              A[i][j] = ...
```

Figure 1: Simple Program

In general, dynamic scheduling algorithms can achieve good load balance, but at the cost of decreased locality in data accesses, since each subtask may be scheduled on any of the processors regardless of the location of the data it must access. The cost of an average memory access would increase on systems with non-uniform memory access cost. This can lead to a decrease in performance, not only because of increased latencies to access the remote data, but also because of increase in network traffic and congestion.

It is interesting to note that, while cache memory helps reduce the effects of non-uniform memory access costs, it does not eliminate them entirely. In practice, even on a multiprocessor with hardware

```
for(k=0; k < N-1; k++) {
        d0 = A[k][k];
        parallel_for(j= k+1; j< N; j++) {
                A[j][k] /= d0;
                d1 = A[j][k];
                for ( i = k+1; i < N; i++)
                        A[j][i] -= d1*A[k][i];
       }
}
```

Figure 2: LU Decomposition

cache consistency (such as the DASH multiprocessor), the average memory response time can be reduced by exploiting memory locality. We illustrate this using LU decomposition as an example. The core of the LU decomposition code is shown in Figure 2. It consists of an outer sequential loop and a parallel loop. In the innermost loop, one of the rows of the matrix A is modified based on the pivot row k. Consider the execution of the parallel loop $j = 5$ running on processor $P1$, and trace the computation. $P1$ accesses the elements of the fifth row and modifies it, causing a valid copy of this row to be in the cache of processor $P1$ and the copy in the memory to become invalid (assuming a write-back cache). If in the next invocation of the parallel loop, loop $j = 5$ is executed on another processor (say $P7$), then the main memory needs to be updated with the values still cached on $P1$, and the values in $P1$ need to be invalidated when $P7$ modifies them. Thus the validation and invalidation traffic on the network can become excessive if there is no locality. Similar effects are possible in cache-only memory architectures (COMA), such as the KSR.

Some multiprocessors, such as RP3 and BBN Butterfly/TC-2000 allow remote memory to be accessed in an interleaved and/or randomized manner. By distributing the memory accesses more evenly across the processors, the number of hot-spots at the memories and in the network is reduced. Thus the NUMA machine behaves like an UMA machine with one cost (close to the maximal one) for accessing the memory. While randomized access has the possible advantage of reducing memory and network hotspots, it also has the disadvantage of not exploiting locality. Because a single processor can execute using only local memory, the speedup of applications that exclusively access shared data in this randomized manner will be poor. The execution time of the application would be better, if data locality were exploited.

4 Locality-based Dynamic Scheduling

In this section, we propose a new scheduling algorithm, Locality-based Dynamic Scheduling (LDS), that addresses both locality and load balancing. LDS (see Figure 3) is based on the following principles:

1. The data space is partitioned to reside on P processors. Typical data partitions are block, cyclic, and block-cyclic. Often, the partition chosen is constrained by rest of the computation.

2. Each processor, when it is ready to execute the next subtask, computes the size of this subtask. The size can be chosen as in any of the dynamic scheduling algorithms as a function of the number of remaining iterations and the number of processors. In our experiments, we set the subtask size to $\lceil n/(2P) \rceil$ (where n is the number of remaining unscheduled iterations and P is the number of processors). This creates subtasks about half as large as those by GSS in order to avoid overly large initial subtask sizes (see Table 1.

3. Once the subtask size has been determined, the processor must decide which iterations to execute as part of its subtask. The dynamic scheduling algorithms sequentially take iterations from the loop iteration space; that is the first subtask of size $S1$ includes iterations $1, 2, \ldots, S1$, the second subtask of size $S2$ includes iterations $S1 + 1, \ldots, S1 + S2$, and so on. In LDS, on the other hand, the iterations are chosen such that locality is maximized. For example, if the data distribution is cyclic, and the processor p has to execute a subtask of $S1$ iterations, then it executes the iterations $p+P, p+2P, \ldots, p+P*S1$. If the data distribution is block then the subtask would include iterations $p*B+1, p*B+2, \ldots, p*B+S1$, where B is the block size. If all the scheduled local iterations are completed, iterations are acquired from the processor with the most unscheduled iterations.

The LDS algorithm is related to the *affinity scheduling* algorithm (AFS) proposed by Markatos and LeBlanc in that AFS also takes locality into account [9]. AFS divides the iterations of a loop into block partitions of $\lceil N/P \rceil$ iterations, where N is the

1. Determine the subtask size $S = \lceil n/(2P) \rceil$ based on the total number of unscheduled iterations (n) and the number of processors (P).

2. If the processor has $r > 0$ locally assigned, unscheduled iterations, then the subtask includes $\min(r, S)$ of those iterations. Otherwise, if $r = 0$ then $\min(r_{max}, S)$ iterations are acquired from the processor with the most unscheduled iterations, where r_{max} is the maximum number of unscheduled iterations on that processor.

3. Execute the subtask.

4. Repeat 1–3 until $n = 0$.

Figure 3: Locality-based Dynamic Scheduling Algorithm

total number of iterations and P is the number of processors, and assigns each partition to a different processor. When a processor becomes free, it takes the next subtask of $1/k$ iterations from its local partition, where k is a parameter of the algorithm that is chosen statically between 2 and P. Once the entire local partition has been executed, the processor determines the processor with the most remaining iterations, and takes fraction $\lceil 1/P \rceil$ of them. The implementation of AFS uses P local locks to protect the local partitions and a global lock to protect the data structure indicating the processor with the most remaining iterations.

LDS is different from AFS in the following ways:

- AFS assumes that data is copied into local storage when first accessed. This can be done by hardware on machines with cache coherence or by the operating system. AFS does not utilize the information of data placement. Instead, it assumes that data accessed in iteration j is likely to be adjacent to the data used in iteration $j+1$, and thus partitions the iteration space into blocks and assigns a block to each processor. Therefore, memory locality can be exploited only when the data is also partitioned and distributed in blocks. LDS, on the other hand, takes data placement into account, by always having the processor first execute those iterations which have the data local to the processor. For this reason, LDS can easily accommodate other data partitioning methods, such as cyclic or block-cyclic.

	Do not consider Locality	Consider Locality
Static	Block Cyclic Block-cyclic	Block-D Cyclic-D Block-cyclic-D
Dynamic	GSS Self Factoring Trapezoid	AFS LDS

Table 3: Comparing the various scheduling algorithms

- In AFS, each processor independently schedules iterations from its local partition using a parameter k. The best value for k will depend on the application and may be difficult to choose. If k is small, then a processor with an exceptionally large proportion of the workload assigned to it could make the size of the first subtask too large so that later dynamic scheduling will not be able to adjust for the load imbalance. The maximum load imbalance in a loop with a linear iteration space in AFS is $\frac{N(P-k)}{P(P-1)k} + 1$ iterations. When k approaches P, on the other hand, the worst-case load-imbalance approaches that of GSS, but at the cost of an increase in the number of synchronization operations by a factor of P, since the total number of lock operations performed by AFS is $O(kP \log \frac{N}{kP}) + O(P \log \frac{N}{P^2})$ [9]. In LDS, the worst-case load-imbalance will be one iteration and the number of synchronization operations will be $O(P \log(N))$.

- AFS uses one way to determine the number of iterations to be taken from a local partition and another way to determine the number of iterations to be taken from other partitions. LDS uses the same algorithm for the both.

Table 3 gives a comparative classification of the scheduling algorithms we have discussed. The differences between AFS and LDS in particular are summarized in Table 4. Differences in performance are shown in the next section.

5 Experimental Results

In this section, we present performance results from experiments involving four benchmark programs: Matrix Multiplication, LU Decomposition, Successive

	AFS	LDS
Locality	block only	any data dist.
Lock Ops	$O(kP\log\frac{N}{kP})+$ $O(P\log\frac{N}{P^2})$	$O(P\log(N))$
Max Imbal.	$\frac{N(P-k)}{P(P-1)k}+1$ iters	one iter.

Table 4: Comparison of AFS and LDS

Over Relaxation, and Transitive Closure.

- Matrix Multiplication: The regular matrix multiplication i-j-k algorithm is parallelized at the outer i loop, multiplying 400×400 matrix of double precision numbers.
- LU Decomposition: This algorithm has a sequential outer loop, a parallel loop and an inner most sequential loop. A matrix of 400×400 double precision numbers is partitioned by row. The computation of LU decomposition is skewed in that the lower rows must be recalculated more frequently than the upper rows; *i.e.*, row 1 is calculated only once, where as elements of row $N-1$ are processed in $N-1$ iterations.
- Successive Over Relaxation: SOR is similar to LU decomposition in that it has a sequential outer loop and a parallel inner loop. Because each processor must access all the neighboring elements of the element being computed, locality plays a major role in obtaining good performance.
- Transitive Closure: Transitive closure has a loop structure similar to LU decomposition. Unlike the previous algorithms, the sequential inner most loop may or may not be executed, and hence the computation is dynamic. The input values determine the variation of iteration execution time. High variance can cause load imbalance. A matrix size of 800×800 integers is processed.

The experiments were performed on Hector, a scalable shared memory multiprocessor [14, 11]. Hector consists of sets of processor-memory pairs connected together by buses, several buses connected together by local rings, and several local rings connected together by a global ring (see Figure 4). Hector provides a single global physical address space; each memory module contains one portion of the global memory. Access time to memory is a function of memory hierarchy.

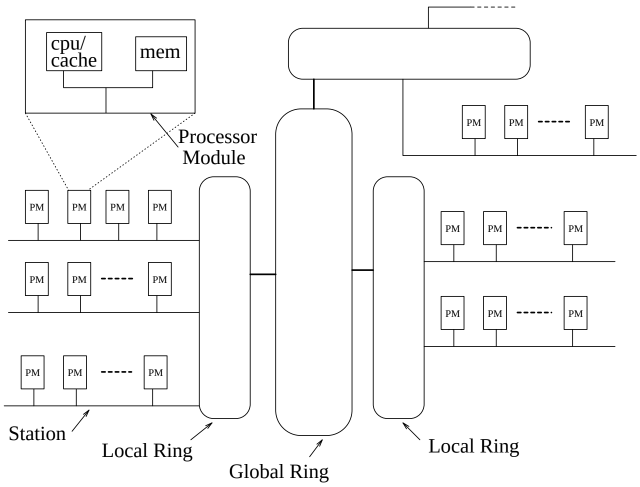

Figure 4: General Architecture of Hector

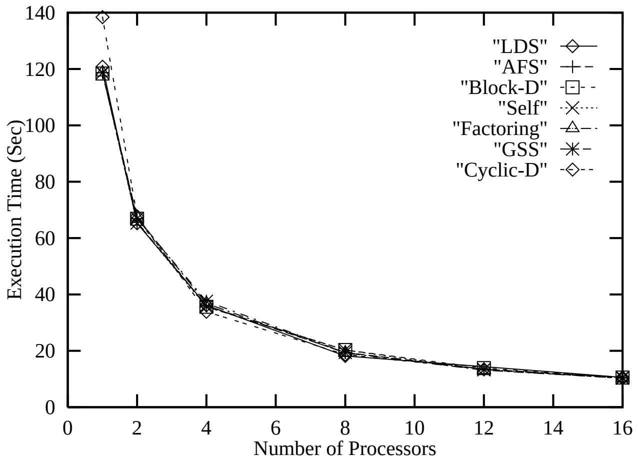

Figure 5: Execution Times for Matrix Multiply

Figures 5-8 show the response times of the four applications listed above when run with the different scheduling policies.

In matrix multiplication, all processes must access all of the matrix B. Assuming B does not fit in the cache, thus the overhead of accessing B will dominate the total overhead of accessing the data elements. Good locality in accessing the elements of matrices A and C is automatically achieved through the caching. For this reason, and because the load in this computation is well balanced, all of the scheduling algorithms perform equally well, as shown in Figure 5. Our results for matrix multiplication differ from those of a similar experiment performed on the RP3 by Hummel *et al.* [4]. In our case, static par-

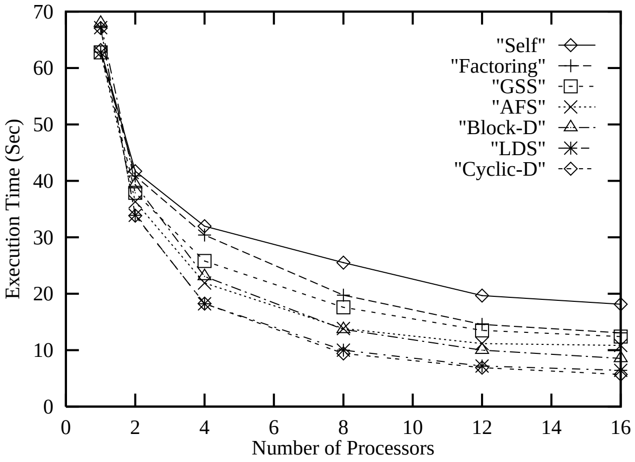

Figure 6: Execution Times for LU Decomposition

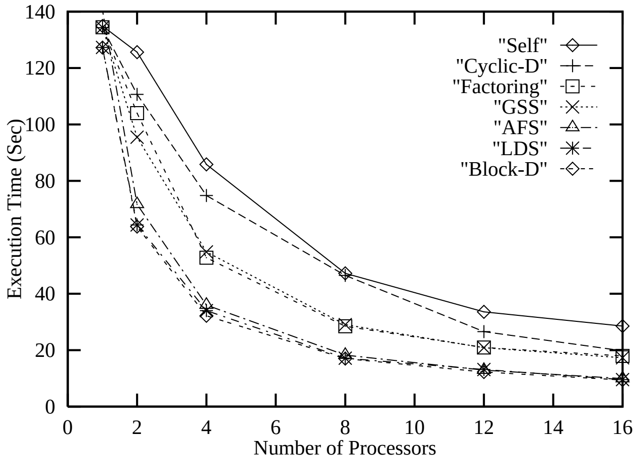

Figure 7: Execution Times for Successive Over Relaxation

titioning, namely cyclic-D, performs marginally better than the other scheduling algorithms. Hummel's results indicate that static partitioning performs far worse than the dynamic schemes. We believe this discrepancy is due to a mismatch between the data partitioning and loop partitioning in the RP3 experiments, making the static algorithm perform poorly.

For LU decomposition, static cyclic scheduling (cyclic-D) outperforms all other scheduling algorithms, because cyclic loop partitioning and scheduling matches the cyclic data partitioning, and balances the load well for the triangular iteration space of LU decomposition. LDS performs almost as well as cyclic-D, and better than the other algorithms for the same reasons. The execution time of the program using LDS is slightly higher than that for cyclic-D because of the run-time scheduling overhead. The results are shown in Figure 6.

For SOR, the static scheduling algorithm that matches the data partitioning, namely block-D, performs best, and the best results are obtained when the data and iteration spaces are partitioned into blocks. Again, LDS performs almost as well as block-D because its iteration space is effectively also partitioned into blocks, given the block distribution of data. In this case affinity scheduling (AFS) performs equally as well because of block partitioning. From Figure 7, it is interesting to note that the performance of the other dynamic algorithms is substantially worse than Block-D, LDS, and AFS because of the lack of data locality.

The transitive closure experiment was chosen as a representative of computationally imbalanced iteration spaces. In this case, LDS outperforms all of the other scheduling algorithms, because it is able to dynamically balance the load, and yet exploit data locality. Block-D performs almost as well, because in this case the variance in the amount of computation of the blocks is not very large. One would expect that AFS could perform as well or better than block-D scheduling, but our results indicate that the overhead AFS incurs for locking will increase quadratically with the number of processors. For example, with $P = 16$, the number of lock operations performed will be about 431, with most locking occurring towards the end of the computation. The deterioration of AFS's performance as the number of processors increases is visible in Figure 8.

Our results indicate that no fixed scheduling algorithm will perform satisfactorily for all applications without taking data locality into account. The static algorithms perform better if the iteration partitions match the data partitioning. With the exception of LDS, the dynamic algorithms are all similar.

6 Conclusions

In this paper we have argued that data locality is an important factor to consider in partitioning and scheduling loops. While most existing dynamic scheduling algorithms manage load imbalances well, they fail to take locality into account and therefore perform poorly on parallel systems with non-uniform memory access times. We have presented a new scheduling algorithm, LDS, that is dynamic, yet takes

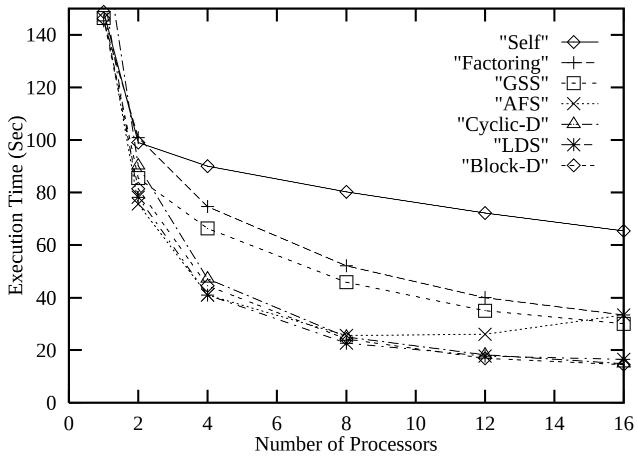

Figure 8: Execution Times for Transitive Closure

locality into account. We have presented experimental results that indicate:

- on NUMA systems, scheduling algorithms do not perform well over a variety of applications if they do not take locality into account;
- no single scheduling algorithm performed best across all applications considered;
- of all the dynamic scheduling algorithms, LDS performed best for the applications considered; and
- unless large load imbalances exist (that is, variance in loop execution times are high), appropriate static algorithms outperform the dynamic scheduling algorithms.

We are currently working on extending LDS to handle irregular data distributions.

References

[1] Gordon Bell. Ultracomputers: A teraflop before its time. *CACM*, 35(8):27–47, August 1992.

[2] William J. Bolosky, Michael L.Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA policies and their relation to memory architecture. In *ASPLOS-IV Proccedings*, pages 212–221, April 1991.

[3] Stephen Curran and Michael Stumm. A comparison of basic CPU scheduling algorithms for multiprocessor Unix. *Computing Systems*, 3(4), Fall 1990.

[4] Susan F. Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *CACM*, 35(8):90–101, August 1992.

[5] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Eng.*, SE-11(10):1001–1016, October 1985.

[6] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[7] Jack G. Lipovski and Miroslaw Malek. *Parallel Computing: Theory and Comparisons*, Appendix C: The RP3. John Wiley and Sons, 1987.

[8] Jack G. Lipovski and Miroslaw Malek. *Parallel Computing: Theory and Comparisons*, Appendix D: Cedar. John Wiley and Sons, 1987.

[9] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Supercomputing 92*, pages 104–113, November 1992.

[10] Constantine Polychronopoulos and David Kuck. Guided self scheduling: A practical scheduling scheme for parallel computers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[11] Michael Stumm, Zvonko G. Vranesic, Ron White, Ron Unrau, and Keith Farkas. Experiences with the Hector multiprocessor. In *International Parallel Processing Symposium*, April 1993.

[12] Arthur Trew and Greg Wilson. *Past, Present, Parallel: A Survey of Available Parallel Computer Systems.* Springer-Verlag, 1991.

[13] Ten H. Tzen and Lionel M. Ni. Dynamic loop-scheduling for shared memory multiprocessors. In *Proceedings International Conference on Parallel Processing*, pages 247–250, August 1991.

[14] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.