How Architecure Evolution influences the Scheduling Discipline used in Shared-Memory Multiprocessors

Evangelos P. Markatos[a]*

[a] Institute of Computer Sci., FORTH P.O.Box 1385, Heraklio, Crete GR-711-10 GREECE
markatos@csi.forth.gr

To appear in Parallel Computing 93.

1. Introduction

Parallel applications execute efficiently, only when they distribute their workload among the available processors, so that *no processors are idle* while there is work to do, and the interactions among the processors in the form of *communication*, or *synchronization* overhead is minimized. Communication is every form of information exchange, including message passing, cache misses and non-local memory accesses. These three overhead dimensions (namely load imbalance, communication and synchronization) are usually in conflict with each other. For example, a policy that balances the load distributes the work (evenly) among processors, thus increasing communication and synchronization overhead. Therefore, the efficient execution of parallel applications relies on the delicate balance among the three overhead dimensions. In this paper we argue that the importance of each dimension, changes with architecture evolution and we study the performance implications of this change.

Load imbalance is the result of an uneven distribution of the work among all available processors, and is inherent to the application and the scheduler used. Architectural changes usually do not change the imbalance as long as the scheduler and the number of processors remain the same. Communication and synchronization overhead, on the other hand, change with the architecture. Actually, this overhead tends to increase as recent architecture trends suggest, since processors are getting faster at a much higher rate than memories and interconnection networks do.

In this paper we investigate the performance implication of this evolving tradeoff for three different scheduling families: static, dynamic and affinity schedulers.

- *Static Schedulers* assign the work to processors at compile time and never reassign (migrate) work to idle processors. Hence, they have very little synchronization/communication overhead, but may lead to underutilization of the multiprocessor.
- *Dynamic Schedulers* make all scheduling decisions at run-time [2]. They use a central work-queue, where all idle processors go to find work to execute. While these schedulers result in minimal load imbalance, they may also result in an increase of communication overhead, because processors execute processes independently of where the working set of these processes may reside. This scheduling discipline may result in large numbers of cache misses or non-local memory accesses.
- *Affinity Schedulers* [1] attempt to strike a balance among the static and dynamic schedulers. To do so, affinity schedulers create one local workqueue per processor. Each processor is statically assigned some work, as if static scheduling were used. If load imbalance actually occurs, idle processors search the workqueues of other processors to find work to do. Thus, affinity schedulers *assign* the work to processors in exactly the same way as static schedulers, but *reassign* the work to idle processors if load imbalance happens.

2. Results

We use simulation and experimental evaluation to measure the performance of the scheduling families. We have chosen transitive closure as a representative application that combines all overhead dimensions and allows us to explore them. The pseudo-code for transitive closure is:

*Part of this work was perfomed when the author was wih the University of Rochester. Part of this work was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, NSF Research Grant No. CCR-9005633, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930).

```
1:      for k = 1 to N
2:         forall i = 1 to N
3:            if (MATRIX(i,k) )
4:                  for j = 0,N,1
5:                      if (MATRIX(k,j)) THEN
6:                           MATRIX(i,j) = TRUE
```

The three overhead dimensions manifest themselves as follows:

- *Synchronization Overhead*: The granularity of the `forall` loop in line `2:` determines the synchronization overhead needed to parallelize the application.
- *Load Imbalance*: Each iteration of the `forall` loop may be long or short depending on the value of `MATRIX(i,k)`. In fact, the input `MATRIX` we use represents a graph which has no edges apart from a clique of size $p \cdot N$, where $0 \leq p \leq 1$. Varying p, varies the imbalance inherent to the application. Hence, a naive assignment of iterations to processors may result in an uneven distribution of the work.
- *Communication Overhead*: The i_{th} iteration of the `forall` loop accesses the i_{th} row of the matrix. If each time the `forall` loop executes, the i_{th} iteration is assigned on a different processor than the one it was assigned the previous time, the i_{th} row will have to be migrated from one local memory (cache) to another.

2.1. Experimental Evaluation

The multiprocessing environment

In our experiments we use two multiprocessors that are representative of their generations and span a time range of over 7 years: (1) The *Sequent Symmetry* (released in 1985) is bus-based cache-coherent multiprocessor with slow processors a rather fast bus, and (2) the *KSR-1* (released in 1991) is a large scale cache-coherent multiprocessor with very fast processors and a large interconnection network. Communication (compared to computation) is much more expensive on the KSR-1, than on the Symmetry.

The Performance of Schedulers on KSR and Sequent

In our experiments we run the application under the three different schedulers and we varied the inherent imbalance by varying p. Fig. 1 shows the completion time of the application on the KSR-1 and Symmetry multiprocessors. The Symmetry results suggest that the dynamic and affinity schedulers are better (almost) everywhere in the range of imbalance. Load balancing seems too important to be ignored on the Symmetry, while communication and synchronization overhead do not manifest themselves.

However, the picture on the KSR is much different. Fig. 1 suggests that when p is more than 80% , static scheduling is the best. This means that even if 20% of the processors are idle, it is better leaving them idle, rather than migrate some work to them. By comparing dynamic and static scheduling only, we see that as much as 50% of the processors should be idle before it is worth migrating some work to balance the load!

Hence, we see that although dynamic scheduling was a reasonable choice for the 1985 Symmetry, it is not a reasonable choice for the 1991 KSR. Static and affinity scheduling are more appropriate. The reason lies in the change of the cost of communication from one multiprocessor to another. As communication is getting more expensive, while load imbalance does not change, communication and synchronization overhead start to manifest itself, making static and affinity schedulers attractive choices.

2.2. Simulation Results

We use simulation to quantify the performance difference among the three scheduler families. Although simulation is only an approximation of the real execution of programs, it is very helpful in answering questions that can not be answered using experimental evaluation. The questions we plan to answer using simulation are:

- What is the performance difference among the scheduler families as we vary the migration cost?
- How much load imbalance in the application is necessary for the dynamic schedulers to outperform the static scheduler? How does this imbalance vary with the number of processors?

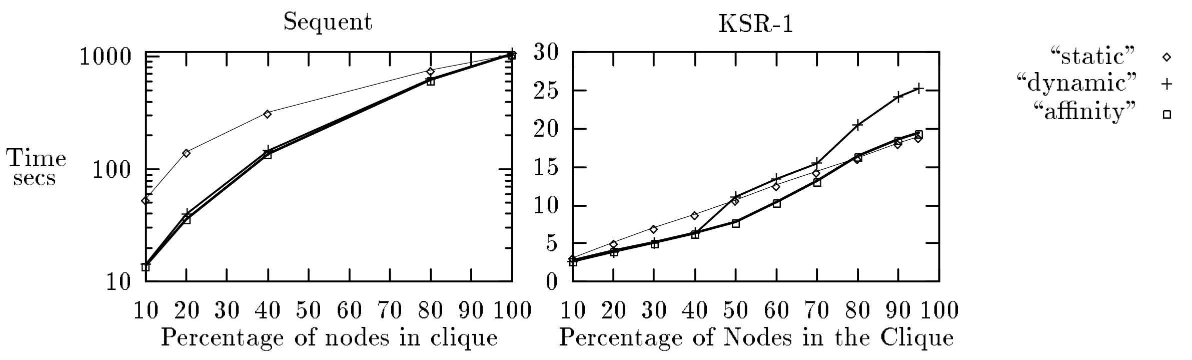

Figure 1. Completion time of TC on Sequent and KSR-1

The simulator

We developed a simulator that simulates the scheduling of N iterations on P processors. Each iteration has affinity for one processor. If the iteration is not executed on the processor it has affinity on, it encounters a multiplicative migration overhead m. The static scheduler assigns each iteration to be executed on the processor it has affinity on. The dynamic scheduler assigns iterations to processors using a self-scheduling [2] method. Each processor takes the next available iteration from the central queue where all iterations reside. The dynamic affinity scheduler assigns each iteration to be executed on the processor it has affinity on. If a processor is idle, it searches the queues of the other processors. If it finds some processor that still has iterations to execute, it takes one iteration, executes it and repeats this cycle until all processors run out of iterations to execute.

2.2.1. Simulation Experiments

Varying the Migration Penalty

In our first set of experiments we vary the migration penalty, that is the penalty an iteration incurs if it is executed on a processor that has no affinity to. For this set of experiments we simulated a 32-processor multiprocessor, that executes 10000 iterations.

Our first experiment assumes an application with $c = 50$, that is, all the load is initially distributed to 50% of the processors, while the rest 50% of the processors took only empty iterations to execute, effectively, they have nothing to do. The completion time of the application under the three schedulers appears in figure 2. The completion time is plotted as a function of the migration penalty. We see that dynamic scheduling is reasonable only when the migration penalty is very small. In this range, static scheduling is the worst of all. When the migration penalty becomes higher than 2, then static scheduling becomes better than dynamic. Affinity scheduling is almost always the best, and is significantly better than the other schedulers only in the range where the migration penalty is close to 2. Everywhere else, affinity scheduling is close to the better of dynamic and static scheduling.

When the imbalance is 90% (figure 2) affinity scheduling seems then only reasonable thing to do. Static is out of the question, and dynamic is somewhere in between.

Varying the Imbalance

In this set of experiments we vary the imbalance the application may have. The imbalance of the application, is represented by the imbalance factor, which is the percentage of idle processors after the assignment of iterations to processors. An imbalance factor of 10 means that 10% of the processors are idle after the initial assignment of work to processors. We perform two experiments: when the migration penalty is 2, and when the migration penalty is 4.

Figure 3 plots the completion time of the application when the migration penalty is 2. We see that when the imbalance is high, dynamic and affinity scheduling are the best, while when the imbalance is low static and affinity scheduling are the reasonable choices. The crossover between static and dynamic scheduling is when the imbalance factor is 50. The next figure plots the performance of the schedulers when the migration penalty is 4. The results are qualitatively the same, only the crossover value has moved to the point where the imbalance factor is about 75. In general affinity scheduling is significantly better than the other policies, only close to the crossover point, while it is close to the best of the policies everywhere else.

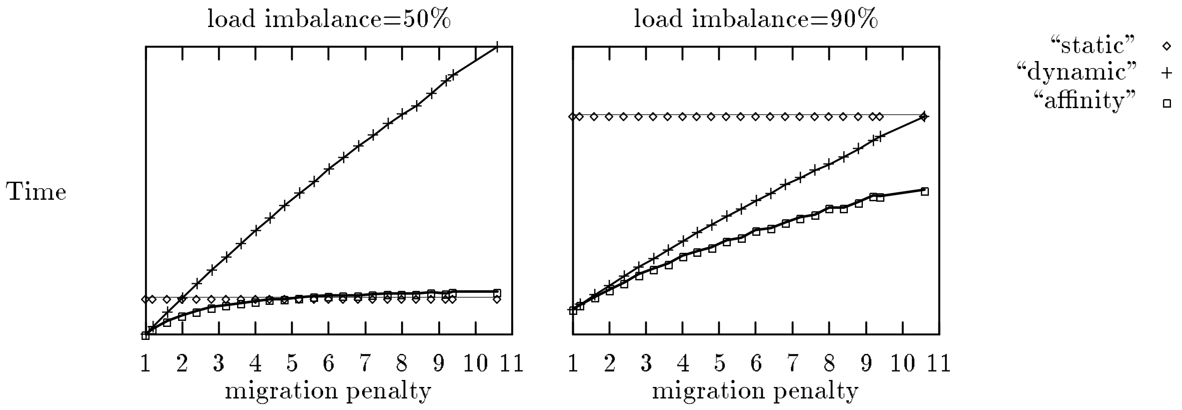

Figure 2. Completion time of the application imbalance factors 50 and 90.

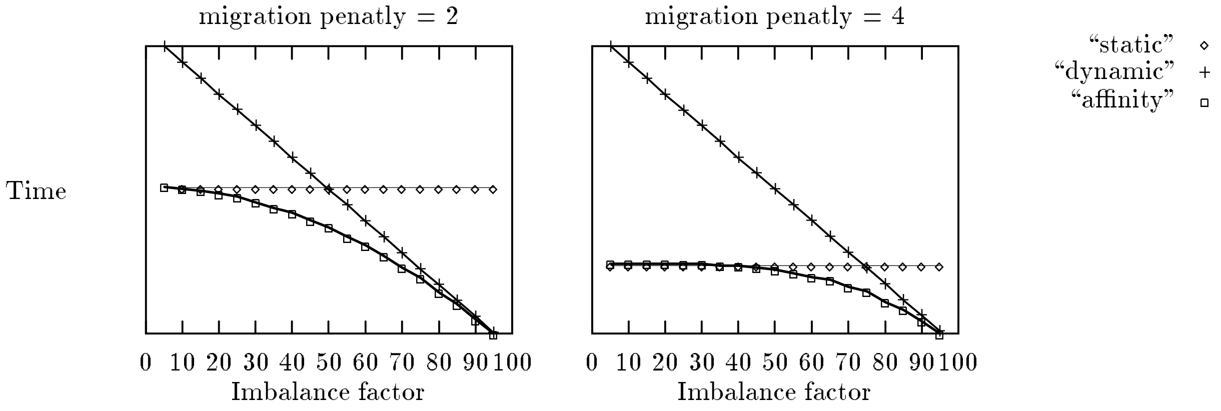

Figure 3. Completion time of the application: migration penalty 2 and 4.

3. Conclusions

In this paper we compared three scheduler families: dynamic, static and affinity schedulers. We used experimental evaluation on the Sequent and KSR shared memory multiprocessors, as well as simulations. Our observations suggest that

- *Dynamic schedulers are appropriate for previous generations of multiprocessors*, but are not appropriate for recent ones.
- *Static schedulers are better than dynamic ones*, when the overhead related to data migration is higher than the overhead related to load imbalance.
- *Affinity schedulers are significantly better than the others only for the range of parameters where the imbalance is comparable to the migration cost*, and close to the best of the other two schedulers everywhere else.

Based on our observations we conclude that the performance advantages of static schedulers (low communication and synchronization overheads) become more apparent with architectural trends, while the advantages of dynamic schedulers (load balancing) do not improve with time. In the presence of little imbalance, static schedulers are the appropriate choice, while in the presence of significant imbalance affinity schedulers should be employed.

REFERENCES

1. E.P. Markatos and T.J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Supercomputing '92*, pages 104–113, November 1992.
2. C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.