Load Balancing Performance of Dynamic Scheduling on NUMA Multiprocessors*

M. D. Durand†
Computer and Information Science Department
University of Pennsylvania

T. Montaut L. Kervella
IRISA
Rennes, France

W. Jalby‡
Laboratoire MASI
Université de Versailles, France

September 9, 1997

Abstract

Self-scheduling is a method for task scheduling in parallel programs, in which each processor acquires a new block of tasks for execution whenever it becomes idle. To get the best performance, the block size must be chosen to balance the scheduling overhead against the load imbalance. To determine the best block size, a better understanding of the role of load imbalance in self-scheduling performance is needed.

In this paper we study the effect of memory contention on task duration distributions and, hence, load balancing in self-scheduling on a Non Uniform Memory Access (NUMA) machine. Experimental studies on a BBN TC2000 are used to reveal the strengths and weaknesses of analytical performance models to predict running time and optimal block size. The models are shown to be very accurate for small block sizes. However, the models fail when the block size is large due to a previously unrecognized source of load imbalance. We extend the analytical models to address this failure. The implications for the construction of compilers and runtime systems are discussed.

Keywords: Dynamic scheduling, load balancing, memory performance, self-scheduling, NUMA multiprocessors.

*Appeared in IEEE Transactionson Parallel and Distributed Systems, Nov. 1996,p. 1201 – 1214

†Durand's contribution to this work was performed in part at IRISA and Ecoles des Mines de Paris and supported by the North Atlantic Treaty Organization under a Grant awarded in 1990.

‡Jalby's contribution to this work was performed in part at IRISA.

1 Introduction

Non-Uniform Memory Access (NUMA) machines are designed to provide the scalability of a distributed memory machine with the ease of programming associated with the shared memory model. Many research and commercial NUMA multiprocessors are currently being designed and built, including RP3 [1], Ultracomputer [2], DASH[3], Hector [4], KSR [5], Cedar[6] and the BBN Butterfly machines [7]. On these machines, each processing element includes a processor and some local memory. The processors are linked by an interconnection network. A global address space allows the programmer to view the physically distinct local memories as a single shared memory. However, because the memories are physically distributed, the performance characteristics are quite different from a true shared memory machine.

In this article, we consider the impact of memory performance on task scheduling on NUMA multiprocessors. Task scheduling may be performed statically or dynamically. To obtain efficient static scheduling, we must be able to predict task duration accurately to avoid load imbalance. As we shall see later in this paper, when memory contention makes a significant contribution to the task duration, task prediction is extremely difficult. Hence, we consider dynamic scheduling methods.

Self-scheduling is a dynamic scheduling technique in which processors autonomously access a global data structure to obtain additional work when they become idle. It is a good technique for balancing the load when task duration is unpredictable. Furthermore, it offers the user a simple programming model. However, the contention and overhead associated with the global task pool can reduce performance. To reduce overhead, a block of K tasks is allocated simultaneously. The basic problem is to choose a block size that is sufficiently large to amortize the scheduling overhead without creating a load imbalance.

In order to use self-scheduling efficiently, we need a better understanding of its behavior on NUMA machines so that we can correctly predict the optimal block size and tune the performance for the architecture and application at hand. In this paper, we offer two major contributions. First, we study the impact of memory performance on task duration distributions experimentally on a BBN TC2000. We demonstrate how the shape and width of the distribution of task durations affects the magnitude of the load imbalance and the choice of optimal block size, K_{opt}. Our results emphasize the importance of using dynamic scheduling on NUMA machines. Second, we show that analytical models of self-scheduling fail in certain regions of the parameter space due to a previously unidentified source of load imbalance. We present an analytical approach of finding those regions where the models are unreliable. A more accurate method for predicting the optimal block size in those regions is presented.

The paper is structured as follows. In Section 2, we review self-scheduling schemes. Analytical models to predict running time and optimal block size are discussed in Section 3. Next we present experimental studies on the TC2000. In Section 4, we describe the experimental approach and present the impact of memory contention and operating systems effects on task durations. In Section 5, we compare the measured running time with the time predicted by the analytical models reviewed in the previous section. We look at how well the models predict the optimal block size, K_{opt}, as well as the quality of the running time predictions. The results here are surprising. The models perform extremely well for small K but fail when K is large. We present an analysis to explain this phenomenon and suggest some solutions. Our results are summarized in the conclusion.

2 Self-Scheduling Schemes

In this section, we review algorithms related to self-scheduling. Analytical models of self-scheduling are reviewed in the next section.

A number of self-scheduling algorithms with decreasing block size have been presented in the literature. These algorithms attempt to balance the load by reducing the size of the blocks allocated as the algorithm progresses. Together with fixed block size self-scheduling, they form a family of algorithms with similar structure and behavior. A better understanding of one member of the family can yield new insights into others. For example, the mathematical analysis that led to the development of the decreasing block size scheme Factoring [8] was based on an analytical model of fixed block size self-scheduling [9].

The problem in designing a decreasing block size self-scheduling algorithm is to determine how large the first block allocated should be and at what rate to reduce the size of subsequent blocks so as to minimize overhead without incurring a load imbalance. In Guided Self-Scheduling (GSS) [10], each time a processor becomes idle, it acquires $1/p$th of the remaining tasks, where p is the number of processors. GSS guarantees that, when task duration is constant, the load imbalance is never greater than a single task duration, thus addressing the case where load imbalance results from varying processor starting times. For some other distributions, particularly those with decreasing task durations, GSS allots too much work at the beginning of the schedule, leaving too few tasks to smooth out unevenness at the end. To address this problem, Trapezoidal Self-Scheduling (TSS) [11] uses a parameterized scheme where the block size decreases linearly rather than exponentially. The size of the first block and the rate of decrease depend on the values of the parameters. TSS is also designed to have faster queue accesses than GSS. A third method, Factoring [8], is based on a mathematical analysis to determine the expected amount of work that can be assigned to each block and still, with high probability, leave enough tasks to smooth over the uneveness in finishing times. The resulting formula for the optimal block size at each step depends on the mean and variance of the task duration so that this method can be adapted to a wide range of applications. A simplified version of Factoring that is independent of μ and σ has been shown to work well in practise. Factoring is the only scheme supported by a comprehensive mathematical model.

In addition, two decreasing block size algorithms have been presented that take data locality into account, Affinity Scheduling [12] (AFS) and Locality-based Dynamic Scheduling [13] (LDS). In both schemes, a subset of the tasks is placed in a local queue on each processor. The data needed by those tasks is stored on the same processor. The processor executes its local tasks using a decreasing block size scheme until they are exhausted. Then, if other processors are still working, it acquires additional tasks from another processor, incurring the cost of accessing the remote data in order to process them. AFS and LDS differ in the rate at which the block size decreases and the method used for acquiring additional work from other processors. AFS supports a blocked data partitioning scheme; that is, one in which contiguous data are accessed in order. Their scheme allows data reuse when a parallel loop is executed repeatedly inside a serial loop. LDS also supports other data partitioning schemes such as cyclic and blocked cyclic partitions, allowing the exploitation of locality in the data.

Initially, decreasing block size schemes were viewed as a panacea to the load imbalance problem: a decreasing block size scheme would free the user from having to readjust the self-scheduling scheme to the attributes of the each new application in order to obtain good performance. However, the experience reported with the decreasing block size schemes surveyed above shows that those schemes must also be tuned to deliver good performance. For example, Hummel et al. showed that to achieve good load balancing with high probability the rate of decrease should depend on the mean and standard deviation of the task duration. Another example is the parameterized nature of TSS. How to choose the best values of the TSS parameters is not well understood.

There is also experimental evidence to suggest that if the block size is chosen correctly, fixed block size self-scheduling can perform comparably to decreasing block size self-scheduling. For example, Hummel et al. [8] and Tzen and Ni [11] compared Factoring and TSS, respectively, with self-scheduling using $K = 1$ ("self scheduling" or SS) and $K = N/p$ ("chunk or static scheduling" or CSS). Although they did not study self-scheduling performance using the optimal block size K_{opt}, since $K = 1$ and $K = \lceil N/p \rceil$ constitute two extremes of a range of possible block sizes, these results are instructive. Hummel et al. show that in three out of four experiments (Gauss-Jordan, reverse adjunct convolution and adjunct convolution) some fixed block self-scheduling method (either SS or CSS) performed comparably to Factoring and in the matrix multiplication experiments SS performed only slightly worse than Factoring. In all experiments, either SS or CSS ran as well as or better than GSS. In [11], one of the fixed block methods (either CSS or SS) performed comparably to TSS and better than GSS in all experiments except one. This suggests that if the optimal block size can be determined quickly and accurately, then fixed block size self-scheduling algorithms can offer comparable performance to decreasing block size algorithms.

Fixed block size self-scheduling offers a number of other advantages. It is easier to implement and easier to analyze theoretically. It has already been implemented in runtime systems on a number of multiprocessors [14, 15, 16]. Commercial compilation systems, such as KSR's PRESTO runtime library [5], often use fixed size blocking. Finally, it is much easier to combine fixed block size self-scheduling algorithms with data locality than decreasing block size schemes. No commercial compiler of which we are aware can handle data locality in multiply nested loops using a decreasing size blocking scheme.

3 Analytical Models

In the previous section, we reviewed self-scheduling schemes based on decreasing block size. In this section we discuss analytical models of self-scheduling for fixed block size. Because they provide a mathematical framework for reasoning about self-scheduling, these models also give insight into the behavior of decreasing block size schemes and influenced Hummel et al. in their mathematical analysis of load imbalance with such schemes. We review two models presented by other researchers and then present an extension which has a more accurate estimate of the load imbalance and takes varying processor starting times into account.

Kruskal and Weiss model a program decomposed into N parallel subtasks. A constant number of processors, p, are applied to this program. As processors become idle, they repeatedly acquire blocks of K tasks for execution.[1] The cost, h, of acquiring a block of tasks is fixed. The task durations are a set of independent, identically distributed (*iid*) random variables $\{D_i,\ i \in 1 \ldots N\}$ with density f, distribution F, mean μ and standard deviation, σ. Furthermore, F must have an increasing failure rate (IFR) and possess a moment generating function. The importance of the IFR hypothesis is discussed in more detail below. The block duration is represented by the random variable, X_i, where $X_i = \sum_{j=1}^{K} D_{ij}$ represents the running time of the ith block. Since the duration of a block is the sum of the durations of its constituent tasks, the distribution function of the block duration is simply the K-fold convolution of the distribution function of the tasks, $G = F^{*K}$.

3.1 Kruskal and Weiss

The earliest and most comprehensive model of fixed-size block scheduling was presented by Kruskal and Weiss[9]. Their analysis is based on splitting the program execution into two distinct phases and estimating the running time of each phase separately. This allows the authors to analyze the body of the computation and the load imbalance at the end separately. The difficult part is the estimation of the load imbalance, which stems from the variance in task duration. Subsequent authors have followed this general approach but have used other techniques for estimating the load imbalance in phase two. In Kruskal and Weiss' analysis, phase one begins when the first processor begins working and ends when the last block is allocated. Phase two begins at the end of phase one and ends when the last processor to finish working terminates. The phases are shown schematically in Figure 1, where T_1 is the time when the last block is allocated and T_2 is the moment when all work is terminated. In this figure, the circles represent the overhead associated with acquiring a block of work and the rectangles represent the running time of the blocks. The final circle for each processor corresponds to the additional access required to verify that the queue is empty.

Kruskal and Weiss give bounds on the expected running time of phase one. Restated in the terminology of this paper, their bounds can be expressed as

$$\left(\frac{N-Kp}{p}\right)\mu + \left\lceil\frac{N}{K}\right\rceil\frac{h}{p} \le E[T_1] \le \frac{\lfloor (N-1)/K \rfloor K}{p}\mu + \frac{\lfloor (N-1)/K \rfloor + p}{p}h\,. \tag{1}$$

The lower bound corresponds to the case where $K \sim N/p$ and almost all work is performed in phase two. The upper bound corresponds to the case where $p-1$ processors have finished working when the last processor takes the last block and almost all the work is performed in phase one.

Estimating the running time of phase two is equivalent to determining the expected time the last processor will finish. This reduces to the problem of estimating $E[\max\{X_i\}]$, the maximum of the set of random variables, X_i. Kruskal and Weiss prove that the *characteristic maximum* closely approximates $E[\max\{X_i\}]$ when p is large and the distribution function of the block duration is IFR. They derive an approximation for the characteristic maximum based on the fact that when K is large, the distribution of the blocks can be approximated by a normal distribution regardless of the distribution of the task duration. This approximation is combined with the results

[1] The last processor to request work may receive an odd-sized block if N is not evenly divisible by K.

in Inequality 1 to yield an estimate of total running time, when p and K are large, $K \gg \log p$ and all processors start at the same time:

$$E[T] \approx \frac{N}{p}\mu + \frac{Nh}{pK} + \sigma\sqrt{2K\ln(p)}\,. \tag{2}$$

This equation is appealing because each term has an intuitive meaning. The first term describes the time required for p processors to execute N tasks in a perfectly efficient system. The second term describes the scheduling overhead. The third term corresponds to the load imbalance which occurs due to the variation in task duration. Equation 2 shows the tradeoff between the overhead and the load imbalance: as K increases, the overhead decreases by a factor of $1/K$, whereas the load imbalance increases as $\sqrt{K}$.

Equation 2 is valid for a wide class of distributions, but is restricted to the case where K and p are large. For the case where K is small, Kruskal and Weiss use renewal theory to develop the following estimates of $E[T]$:

$$E[T] \approx \frac{N}{p}\mu + \frac{Nh}{pK} + \sigma\sqrt{2K\ln(\frac{p\sigma}{\sqrt{K}\mu})}\,,\ K \ll N/p \text{ and } \sqrt{K}/p \text{ is small.} \tag{3}$$

$$E[T] \approx \frac{N}{p}\mu + \frac{Nh}{pK} + \frac{p\sigma^2}{\mu}\,,\ K \ll N/p \text{ and } \sqrt{K}/p \text{ is large.} \tag{4}$$

The models discussed above all require that the task distribution have an increasing failure rate. The IFR hypothesis is used to derive the lower bound in Inequality 1 and to estimate the load imbalance. A distribution, F, is IFR if $\frac{1-F(x+t)}{1-F(x)}$ is monotone decreasing for all $t > 0$. Furthermore, $G = F^{*K}$ is IFR if F is IFR. Hence, it is sufficient to verify that the task duration is IFR to ensure that the block duration is IFR. Intuitively, when the task distribution is IFR, the longer the task has been running, the higher the probability that it is about to complete. Most common distributions, such as the truncated normal, gamma and uniform distributions, satisfy these restrictions, but some, such as the Bernoulli distribution, do not. In particular, no bimodal distribution is IFR. This is unfortunate since bimodal task distributions frequently occur. The Bernoulli distribution, for example, is often used to model the task duration of an if-statement with branches of uneven length. As we shall see in the experimental section, bimodal distributions arise for other reasons as well.

Kruskal and Weiss tested their model using simulations based on the normal, exponential, Bernoulli, uniform and polynomial distributions for the task duration. Their model gave good runtime predictions except with the polynomial distribution, which does not have a moment generating function. It is interesting to note that they obtained good results for the Bernoulli distribution, despite the fact that it is not IFR. This suggests that a weaker initial hypothesis may be sufficient. Based on their simulation results, Kruskal and Weiss drew the conclusion that blocks of size $K \sim N/p$ gives good performance. We shall see later in the paper that this is sometimes but not always true.

Kruskal and Weiss estimate optimal block size by finding the value of K which minimizes Equation 2. Taking the derivative and setting it equal to zero yields

$$K_{opt}^{KW} = \left(\frac{\sqrt{2}Nh}{\sigma p\sqrt{\ln(p)}}\right)^{2/3}. \tag{5}$$

Unfortunately, the original equation is only valid when K is large. Hence, the value of K derived from Equation 5, K_{opt}^{KW}, will only be valid if the true optimal block size, K_{opt}^{true}, is large. Analytically tractable expressions for K_{opt} cannot be derived from Equations 3 and 4.

3.2 Madala and Sinclair

A similar approach to estimating the running time was presented by Madala and Sinclair [17]. Instead of the characteristic maximum, they used a distribution-independent bound, $E[\max\{X_i\}] \le \mu + \frac{p-1}{\sqrt{2p-1}}\sigma$, to estimate the duration of the load imbalance [18]. This expression has the advantage of simplicity but is less accurate an estimate of load imbalance for large values of p. Translated into the notation used in the current paper, their expression yields

$$E[T] \leq \frac{N}{p}\mu + \frac{Nh}{pK} + \sigma\sqrt{K}\frac{p-2}{\sqrt{2p-3}} + K\mu + h\,. \tag{6}$$

The first two terms, corresponding to the perfectly efficient running time and the overhead, are identical to those given by Kruskal and Weiss in Equation 2. The third term represents the load imbalance. The final two terms represent the running time required to allocate and execute a single block of tasks. These two terms appear because, in their analysis, Madala and Sinclair present an upper bound. Madala and Sinclair also discussed the problem of optimal block size. However, because they did not present an analytically soluble expression for block size we were not able to use their method in this study.

3.3 A New Extension

The success of the analytical models above depends on how well they estimate the load imbalance. Kruskal and Weiss' estimate is quite accurate but requires three different equations to cover the parameter space. Madala and Sinclair use a much cruder estimate which is, however, simpler and valid everywhere. To obtain an accurate estimate that is valid everywhere, we present a refined version of Kruskal and Weiss' estimate using order statistics to compute the load imbalance more precisely.

Let $\{X_1 \cdots X_p\}$ be a set of iid, random variables with a common distribution function, G, and density function, g. These random variables can be permuted so that they are in increasing order, yielding a new set of random variables, $\{X_{(1)} \cdots X_{(p)}\}$ called the *order statistics* of $\{X_1 \cdots X_p\}$. In particular, $X_{(p)}$ is the maximum of $\{X_1 \cdots X_p\}$. Its expectation, $E_p[G]$, can be expressed mathematically as

$$E_p[G] = E[X_{(p)}] = p\int_{-\infty}^{+\infty} x[G(x)]^{p-1}g(x)dx. \tag{7}$$

If the random variable, X_i, represents the running time of the block on processor i, then $E[X_{(p)}]$ is the expected finishing time of the last processor to finish. Remember that the distribution function of the block duration is simply the K-fold convolution of the distribution function of the tasks, $G = F^{*K}$. This expression can be used to estimate the load imbalance, yielding a new estimate of the total running time

$$E[T] \simeq E[T_i] + (\frac{N-Kp}{p})\mu + \left\lceil\frac{N}{K}\right\rceil\frac{h}{p} + E_p[F^{*K}]\,, \tag{8}$$

as well as an upper bound

$$E[T] \leq E[T_i] + \frac{\lfloor(N-1)/K\rfloor K}{p}\mu + \frac{\lfloor(N-1)/K\rfloor + p}{p}h + E_p[F^{*K}]\,. \tag{9}$$

A complete development of these equations can be found in [19]. The first term on the right hand side is the expected processor starting time, where T_i is the time processor i starts working. The assumption that all processors start at the same time is not realistic since the serial nature of the task queue can cause processors to be delayed in acquiring their first task. This is confirmed in the experimental section of the paper. In addition, processors may be working on other jobs at the beginning of the parallel section of the code. An additional term that takes varying processor starting times into account is needed. This fact is not taken into account in either model surveyed.

The estimate of the load imbalance in the fourth term is valid everywhere and is more accurate than the distribution-free bound used in Equation 6. Analytical expressions exist for the convolution of a number of classical distribution functions such as the normal, gamma and exponential distributions. When an analytical expression for $E_p[F^{*K}]$ does not exist, it can be computed numerically. Although more difficult to compute than the previous models, Equation 8 allows us to gauge the sensitivity of running time estimates to inaccuracies in the computation of the load imbalance.

4 Impact of Memory Latency on Task Duration

In this section we present an experimental study of the effects of memory contention on task duration on a BBN TC2000. As seen in the previous section, the expected running time and the choice of the optimal block size, K_{opt}, depend on the task duration distribution. A number of factors can cause variable task duration: memory contention, if-statements, dynamic data structures and algorithmic constructs. Memory contention is of prime importance for two reasons. First, the impact of memory contention on task duration is extremely difficult to predict analytically. Second, on NUMA machines, the effects of memory contention will interact with on all other factors.

4.1 The Target Architecture

The BBN TC2000 is a MIMD multiprocessor with a hybrid shared/distributed memory structure [7]. Individual processor boards are connected by a nonblocking multi-stage interconnection network. The global memory is distributed across the boards, but has a single address space. Each board contains 16 Mbytes of memory, a Motorola 88100 processor, a 16 Kbyte instruction cache, a 16 Kbyte data cache, switch interface circuitry and control circuitry(see Figure 2). The components on the board are connected by a T-bus. Thus the CPU can access data stored on its local memory via the T-bus. Data references to distant memory locations are decoded by the switch interface circuitry and sent across the switch. Logically, the programmer can ignore the difference between global and local memory. However, global accesses are slower than local ones.

Global memory accesses are performed by establishing a connection between the requesting board and the server board. Once this connection is established it is held open until the operation is complete. In order to establish a connection, the requesting processor must reserve an output port in each stage of the network. There is no buffering in the switch components. If a port is unavailable, it relinquishes all previous ports and retries after a random delay. After repeated failures, the length of the delay is increased using a random exponential backoff strategy.

Cache coherency has not been implemented for the data cache on the TC2000. It is the programmer's responsibility to implement cache coherency in software. Preliminary experiments [20] suggest that for this reason, use of the data cache is not profitable except large matrices or for read-only applications.

A number of software packages for parallel programming support are available on the TC2000. The *Uniform System* (US) supports memory interleaving and self-scheduling. To use the self-scheduling capability, the programmer provides the task code and specifies the priority of the tasks. The US automatically manages the acquisition of tasks by processors in the correct order. The *Gist* performance tool supports the generation and visualization of event traces. Timestamped events are logged in local buffers at a cost of approximately 6 microseconds when implemented as C macros and approximately 8 microseconds when implemented as a function call. The local event buffers are merged at the end of program execution to yield a global event trace. Timestamps are based on the local real time clock on each processor. The clocks are synchronized in hardware and no evidence of clock drift was observed.

4.2 Experiments Performed

We developed a comprehensive test suite allowing us to explore the memory contention parameter space extensively. Experiments were run using synthetic task kernels designed to test the sensitivity of different types of instructions to the network load as well as the propensity of these instructions to create delays. The variable parameters of the test suite were

- the memory storage strategy
- the ratio of global to local memory accesses
- the ratio of the overhead to the task duration
- the number of processors.

Because of the lack of hardware support for cache coherency, cache behavior was not studied.

Varying the ratio of global to local memory accesses allows us to study tasks with varying degrees of global network activity. Earlier studies [21, 22] suggest that local memory accesses are insensitive to network activity. Hence, local memory operations can be treated in the same way as computational instructions. Two types of memory storage strategy were used, *interleaved* and *hotspot*. These storage schemes are two extreme ends of a range. Real programs will probably contain a mixture of these. In the interleaved strategy, the rows of a matrix are evenly distributed across the memories on the processing units working on the current problem. This is done automatically by the US library. In the hotspot strategy, the entire global data structure (in our case a two-dimensional array) is stored in a single global memory.

The source code for our synthetic task is shown in Figure 3. Each task has a global section and a local section. Event logging instructions are used to mark the beginning and end of the task as well as to separate the two sections. The arrays G and B were stored in global memory (in a single memory for hotspot experiments and evenly divided over all processors for interleaving experiments.) Each processor has its own private variable, a, stored in its local memory. The variables $gmax$ and $lmax$ are the loop bounds of the global and local sections respectively. The ratio of global to local memory accesses is controlled by the relative values of $gmax$ and $lmax$. The choice of $gmax$ and $lmax$ also determines the task execution time and what percentage of that time is spent performing memory operations across the network.

Because the ratios of h to μ and σ affect the predictive behavior of the models, experiments were performed with both small and large overheads. The overhead is the elapsed time between the completion of one task and the beginning of the next task on the same processor. The small overhead used was the minimal overhead associated with the US self-scheduling system; that is, the time required to access the global data structure associated with the queue and execute the code which allocates the next task using that data structure. The large overhead is simulated by executing a busy-wait loop at the end of each task. Simulating a large overhead permits us to consider a range of overhead to task granularity ratios.

Experimental parameters are shown in Table 7 and the ensemble of experiments performed is shown in Table 2. For each of the 60 combinations of parameters shown, we ran a program consisting of $N = 50p$ identical tasks and measured the distribution of task durations as well as overhead, processor starting times and total running time. Because of the non-deterministic nature of parallel machines, each experiment was run five times. All tables presented below contain averages of five values.

4.3 Experimental Results

Tables 3 and 4 show the average and the standard deviation of task duration for each experiment, as well as the average duration of the local and global portions of the task, the percentage of the task duration spent accessing global memory, the overhead, h, and the expected processor starting time, $E[T_i]$. The impact of contention on task duration can be seen from these tables, where contention increases when either p or $gmax$ increases. In the interleaving experiments the role of memory contention is mild but not negligible. The task duration varies very little with the number of processors, p, for a fixed value of $gmax$ and that the global task duration does not grow faster than the loop bound, $gmax$. However, for high contention ($gmax = 320$), task duration is somewhat sensitive to p. In hotspot experiments, we see that global memory accesses are much more sensitive to network activity. In contrast to the interleaving experiments, the task duration increases with p and exhibits large standard deviations. This large variance is related to the random delay in the switch protocol. The insensitivity of the local task duration to p in both classes of experiments confirms that local accesses are insensitive to global activity except when the contention is very high (Hh40.b, Hh50.b).

Histograms of task duration are shown in Figures 4 and 5. Figure 4 shows task duration distribution for an interleaving experiment with low contention. The secondary and tertiary peaks to the right are due to clock interrupts. On the TC2000, a hard clock interrupt with duration $\sim$100 microseconds occurs every $\sim$10 milliseconds and a soft clock interrupt with duration $\sim$200 microseconds occurs every $\sim$100 milliseconds. For this experiment, the task has a duration of approximately 2550 microseconds (the primary peak on the left). The smaller middle peak corresponds to tasks during which a hard clock interrupt occurred. The tiny peak on the right corresponds to those tasks which were hit by a soft clock interrupt. The peaks are separated by intervals of approximately 100 microseconds as expected. The percentage of tasks to be hit by a clock interrupt is roughly proportional to ratio of the task duration to the interval between clock interrupts (in this case 2550:10000 or approximately one in four). Because interleaved storage causes little contention, the primary peak in this figure is quite narrow. Although all interleaving histograms exhibited this general shape, as $gmax$ and p increase the peaks grow wider and move to the right.

The results of a hotspot experiment are shown in Figure 5. Here we see two separate peaks with distinctly different shapes. This phenomenon is due to a lack of symmetry in the roles of the processors. The narrow peak on the left corresponds to tasks that were executed on $P0$, the processor where the global data is stored. $P0$ has fast access to the data via the T-bus resulting in much shorter and constant task duration. The other processors must access data through the switch which results in longer and more variable memory latencies. These latencies can be seen in the wide, asymmetric bell curve on the right. Notice that this bimodal effect does not occur for interleaved data structures because on the TC2000 all interleaved memory references pass through the switch, even when the data is stored on the same board as the requesting processor. All hotspot experiments have this general shape. As the contention increases, the second peak becomes wider and flatter and the gap between the two peaks increases.

The results of this section show that task duration distributions are not regular. Memory contention causes distributions to widen into bell-shaped curves. In addition, systems effects (e.g. clock interrupts) cause distributions to split into distinct subdistributions. The width of the distribution is highly dependent on the memory storage strategy and the degree of global activity. Since our tasks are all identical, we would expect to see a nearly constant task duration on a pure shared memory machine. On the TC2000 we see that changing the memory access parameters can change the average duration by as much as a factor of ten! (Compare Il10.b with Hh50.b.) Figure 5 shows that within a single experiment, the task duration can vary from 1000 to 9000 microseconds. Furthermore, none of the histograms seen in these experiments represent IFR distributions since all of them have at least two peaks.

From these results we draw two major conclusions. First, task duration on NUMA machines depends on a complex set of architectural parameters and is unpredictable. Although the specific shape of the histograms presented here depends on the TC2000 architecture, our conclusions will be true for a wide class of NUMA architectures. A complex interconnection network will engender a broad, unpredictable distribution and system effects (e.g. interrupts, page faults, TLB misses) will certainly be present. This confirms our hypothesis that dynamic scheduling is imperative on NUMA machines. Second, to effectively model self-scheduling on NUMA machines, analyses must be based on very general models of task duration. Furthermore, those variable block size algorithms which are based on a general model of task duration, such as Factoring, are more likely to succeed on NUMA machines.

5 Comparison of Scheduling Models

In the previous section, we saw that memory contention results in highly variable task durations. In this section we will use our experimental results to demonstrate the strengths and weaknesses of the analytical models in predicting running time for self-scheduling and block scheduling on NUMA machines. We will discuss how to determine the optimal block size and why these models fail in some parts of the parameter space.

5.1 Self-scheduling

Using the measured values of μ, σ, h and $E[T_i]$ reported in the previous section, we can compute the running times predicted by the analytical models discussed in Section 3 and compared them to the true running time. We used all three of Kruskal and Weiss' equations despite the fact that they are valid in different regions of the parameter space. This gives a sense of the sensitivity of $E[T]$ to the quality of the load imbalance estimate.

Actual and predicted running times are compared in Figures 6 - 9. These estimates were computed from Equations 2 - 4, 6 and 8 using $N = 50p$ and $K = 1$. The values used for h and $E[T_i]$ for each experiment are shown in Tables 3 and 4. To compute the fourth term of Equation 8, we determined the probability distribution of the task duration by fitting the histograms presented in the previous section to classical distributions using the maximum likelihood method as described in [23]. We tried fitting several distributions to the data for each experiment. The best results were obtained using the exponential distribution to model data from interleaving experiments. Hotspot data were best modeled using a discrete distribution with two peaks. When necessary, the $E_p[F^{*K}]$ term was computed numerically using MatLab. We computed 99% confidence intervals for all experiments. Error bars were not included in the figures to avoid clutter. The confidence intervals were never more than 5% of the total running time and usually less than 2%.

As can be seen in the figures, the models predict the running time quite accurately.[2] In the hotspot experiments, the first two estimates of Kruskal and Weiss (Equations 2 and 3) and our estimate gave the closest values, with predicted running time within 1% of true running time and often better. Madala and Sinclair's estimate was not quite as good, with predicted running times differing from true running times by as much as 6% in the worst cases. Because it is an upper bound, Equation 6 has a tendency to overestimate the running time. Madala and Sinclair's worst performance occurs when the contention is high. This is because the bound they used to estimate load imbalance gives the best approximation when the underlying distribution is a high, narrow exponential [18]. As the contention increases, the distribution becomes lower and flatter, resembling a high, narrow exponential less and less. Kruskal and Weiss' third estimate (Equation 4) does quite badly with estimates inaccurate by as much as 37%. This is not surprising since Equation 4 is based on the hypothesis that $\sqrt{K}/p$ is large, which is not true when $K = 1$. Furthermore, the load imbalance term in Equation 4 is proportional to σ^2. Hence, it performs worst when the contention is high and σ is large.

In the interleaving experiments, the estimate of Madala and Sinclair is much more successful, with their estimate and our estimate both generating predictions on the order of 1% of the true running time. Kruskal and Weiss' Equations 2 and 3 give results on the order of 2% of true running time. Figure 9 shows the importance of the $E[T_i]$ term. Notice, for example, that in the 50 processor case on all three curves, the true running time ($\diamond$) exceeds Madala and Sinclair's estimate ($\circ$) despite the fact that the latter is supposed to be an upper bound. This is because $E[T_i]$, which is quite large for the 50 processor experiments (see Table 4), is not taken into account in Madala and Sinclair's equation. In contrast, the curve based on our Equation 8 ($\square$), which includes the $E[T_i]$ term, parallels the true running time. This effect is more obvious in the interleaving experiments that in the hotspot experiments. Since the contention is lower in this case, the $E[T_i]$ term is much more significant.

As discussed above, each term in Kruskal and Weiss' model has an intuitive meaning: the ideal running time, the overhead and the load imbalance. In different regions of the parameter space, different terms will dominate so that a model may be accurate in one region and not another. The results in this section show that this approach to modeling self-scheduling works well when $K = 1$. To our knowledge, this is the first time this approach has been tested on actual multiprocessor data. Both Kruskal and Weiss and Madala and Sinclair tested their work using simulations. We obtain slightly better results using the more accurate estimate of the load imbalance obtained with order statistics, but Kruskal and Weiss' estimates also give good predictions (with in 3% of actual run times) and are easier to compute. Furthermore, it is easy to improve Kruskal and Weiss' equations by adding a $E[T_i]$ term. In the next section, we will look at how well Kruskal and Weiss' approach performs when K is greater than one.

5.2 Block Scheduling

In the previous section we considered the performance of self-scheduling when the block size is one. In this part of the parameter space, the overhead dominates the load imbalance in contributing to long running times. The performance can be improved by scheduling more than one task at a time. The problem is to determine the optimal block size, K_{opt}.

We tested the approach of Kruskal and Weiss discussed in Section 3 by determining K_{opt}^{KW} from Equation 5 using experimentally measured values of h and σ. We then re-executed the same suite of experiments, allocating work in blocks of $K = K_{opt}^{KW}$ tasks. In addition, we determined K_{opt}^{true} by measuring the running time experimentally for values of K between 1 and 50 and choosing the value of K which gave the shortest running time.

[2] Remember, since the number of tasks increases with the number of processors, we do not expect the running time to decrease as p increases.

The block scheduling results are reported in Table 5. To understand these results, it is helpful to remember that $K_{opt} \propto (\frac{h}{\sigma} \cdot \frac{N}{p})^{2/3}$. Since $\frac{N}{p}$ is fixed in our experiments, we focus on the ratio $\frac{h}{\sigma}$.

In the hotspot experiments, the standard deviation, σ, is large and hence, K_{opt}^{KW} is small. In the case of small overhead, where h is small as well, Equation 5 yields $K_{opt}^{KW} = 1$. When we plotted $E[T]$ versus K using Equation 8 (see, for example, Figure 10), the minimum estimated runing time also occurs when $K = 1$. Hence we did not run experiments to determine K_{opt}^{true} in this case. For large h, with the exception of experiment Hl10.b, block scheduling led to substantial performance improvement, running between 20% to 25% faster for most experiments. The value of K_{opt}^{KW} differed slightly from K_{opt}^{true} but the performance improvement was almost as good.

The interleaving experiments are quite a different story. Optimal blocking gave modest performance improvement for small overheads and up to 30% improvement for large values of h. However, using $K = K_{opt}^{KW}$, approximately half of these experiments ran *slower* with the "optimal" block size than with self-scheduling. To understand this surprising phenomenon, consider Figure 11, which shows running time as a function of block size for experiment Im30.b. The lower two curves are the expected running times given by Equations 8 and 2. The upper curve is the upper bound on the running time given in Equation 9. The jagged line in the middle shows experimentally measured running times for values of K between 1 and 50.

As the figure shows, when K is large, the running time is highly sensitive to the relative values of N, K and p. If N/K is not an exact multiple of p, some processors will have to execute an extra block, resulting in a large load imbalance. This is why the running time for $K = 45$, which does not divide N evenly, approaches the upper bound, whereas the running time for $K = 50$ is close to the estimates given by Equations 8 and 2. Kruskal and Weiss' model does not take this type of load imbalance into account. It is designed to address load imbalance problems associated with task duration variance. In the region where K is large, a small change in K can lead to a major change in the running time. Hence, a small error in selecting K_{opt} can be disastrous. For example, if we choose $K = 47$ instead of $K = 50$, the program will run 78% slower!

When does the model predict correctly? When K is small, the estimate and the upper bound are close together and Equation 2 gives good predictions. To understand the situation for large K, look at Figure 12, which shows the amount of work assigned to each processor for various values of K. Equation 2 predicts running time accurately when $\frac{N}{Kp}$ is an integer. In this case, each processor receives exactly the same number of blocks, as shown in Figure 12(a). There are additional points (such as $K = 17$ in our example), where the estimate is pretty good even though Kp does not divide N evenly. At these points, most processors are assigned an equal number of blocks giving a fairly even load distribution. A few processors do less work. For example, when $K = 17$, $p = 30$ and $N = 1500$, twenty nine processors are allocated three blocks and only one processor receives two blocks (see Figure 12(b).) As Figure 11 shows, in this case the load is sufficiently well balanced to give a good running time. The worst case occurs when most processor do the same amount of work but a few execute an extra block (Figure 12(c).) For example, when $K = 12$, five processors execute five blocks and the remaining twenty five execute four blocks leading to a large load imbalance and poor running time.

To make this more precise, we define P_- to be the number of processors that execute one fewer block. (If Kp does not divide N evenly, then some processors execute one more block and some execute one less.)

$$\begin{aligned} P_- &= \begin{cases} 0 & N_b \bmod p = 0 \\ p - (N_b \bmod p) & \text{otherwise} \end{cases} \\ &= p\left(1 + \left\lfloor \frac{(Nb-1)}{p} \right\rfloor\right) - N_b, \end{aligned} \tag{10}$$

where

$$N_b = \left\lceil \frac{N}{p} \right\rceil$$

is the total number of blocks. For convenience, we define P_- to be zero when Kp divides N exactly. The best running times are achieved when P_- is a minimum. For example, it is easy to verify that, when $p = 30$ and $N = 1500$, the minimum values of P_- occur at the same points as the minima in Figure 11. These points correspond to situations like those depicted in Figures 12(a) and 12(b). The load imbalance increases as P_- grows.

Using Equation 10, gives us a solution to the problem of computing K_{opt} analytically. As can be seen from Table 5, although Equation 5 does not always choose a value of K which improves performance, it generally selects a value in the right region of the parameter space. Hence, we can choose an approximate value of K_{opt}^{KW} using Equation 5 and then refine our choice by finding the closest value of K to K_{opt}^{KW} such that P_- is minimum. Another solution is to use a decreasing block size algorithm such as those reviewed in Section 2. However, it is important to keep in mind that to be effective over a wide range of applications and architectures, such a method should take μ and σ into account as Factoring does. A decreasing block size method that adjusts to μ and σ at every step may have a substantially higher overhead than block scheduling.

6 Conclusion

In this paper, we studied the performance of blocked self-scheduling algorithms on NUMA machines. Comprehensive experiments exploring a large part of the parameter space were performed on a BBN TC2000. Two causes of load imbalance were identified: high task duration variance due to memory contention and the relative values of the number of tasks, N, the block size, K, and the number of processors, p.

Analytical models to predict running time and optimal block size, K_{opt}, were surveyed and a new extension was presented. The role of task duration distributions in these models was discussed in light of the results of experimental measurements. Memory contention and systems effects were seen to have a significant impact on the shape and width of the task duration distributions. From these results, we conclude that predicting the duration of individual tasks is extremely difficult on these machines. Hence, dynamic scheduling techniques are imperative on NUMA architectures. Furthermore, analytical performance models must be based on very general hypotheses in order to be accurate and robust.

The total running time for each experiment was also measured. These experimental times were compared with the running times predicted by the analytical models. For self-scheduling ($K = 1$), the results were excellent. The best models gave very accurate predictions (differing $\approx 2\%$ from the true value) despite the fact that some initial hypotheses were violated. To our knowledge, this is the first time these models have been tested on real multiprocessor data.

The results for blocked scheduling ($K > 1$) were another story. The models performed well for small K. The running time predictions in this range are accurate. The optimal value of K predicted by the models is not the true optimal but it is close and gives performance improvements up to 30%.

However, the models fail when K is large. Running time prediction is not accurate and a poor choice of K can cause the program to run slower! This is due to the fact that when Kp does not divide N evenly, a large load imbalance may occur because the work is poorly distributed across the processors. None of the analytical models take this type of load imbalance into account. We present analytical results to explain this failure and suggest a more accurate way to choose K_{opt}.

These results are important for the construction of compilers and runtime systems on NUMA machines. As suggested above, on machines with complex memory hierarchies, it is extremely difficult to predict the running times of *individual tasks*. However, given static analysis of the task structure combined with a lookup table of machine dependent information, it is possible to predict *the mean and variance of the task duration distribution* (see [24, 25] for examples.) A new lookup table must be generated experimentally for each architecture, but constructing the table would be a one-time cost. The work of Gallivan et al. [26] on predicting memory latency using load/store kernels suggests how such a table might be constructed. Using the previously published analytical models with the extensions suggested in this paper, the predicted values of μ and σ can be used to predict K_{opt} automatically. Thus, a low cost load balancing scheme can be achieved in the compiler. A 30% performance improvement in the entire workload can substantially improve the throughput on the entire machine.

Our results are important for the development of algorithms with decreasing block size as well. As seen in the work of Hummel et al. [8], an understanding of the variance of task durations on NUMA machines is necessary to determine the optimal rate of decrease.

7 Acknowledgments

We wish to thank to Christine Fricker, Susan Flynn Hummel, Annie Morin, Philippe Robert, Gerardo Rubino and Paul Tukey for helpful discussions. Sarah Long, Bob Thomas and Guy Fedorkow of BBN were very helpful in providing additional information about the TC2000 architecture and software. Michel Dayde and Cerfacs, Dave Levine and Argonne National Laboratories and Eugene Brooks and Linda Woods at Lawrence Livermore National Laboratories made their TC2000s available to us and we are particularly grateful for their support. Special thanks also to an anonymous reviewer who gave particularly detailed and helpful comments. Finally, Dannie Durand wishes to thank the National Science Foundation and Ecoles des Mines de Paris for financial support.

References

[1] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.," in *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 764–771, August 1985.

[2] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. M. e, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, pp. 175–189, February 1983.

[3] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the Seventeenth International Symposium on Computer Architecture*, (Seattle, Washington), pp. 148–159, May 1990.

[4] D. M. L. Z.G. Vranesic, M. Stumm and R. White, "Hector: A Hierarchically Structured Shared Memory Multiprocessor," *Computer*, vol. 24, pp. 73–82, Jan. 1991.

[5] "KSR Parallel Programming," 1991.

[6] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar – A large Scale Multiprocessor," in *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 524–9, Aug. 1983.

[7] "Inside the TC2000," 1990.

[8] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," in *1991 Supercomputing Conference*, pp. 610–619, Nov. 1991.

[9] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1001–16, Oct. 1985.

[10] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, vol. 36, no. 12, pp. 1425–39, 1987. Special Issue on Supercomputing.

[11] T. H. Tzen and L. M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 87–98, Jan. 1993.

[12] E. Markatos and T. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," in *Proceedings of the 1992 Supercomputing Conference*, (Minneapolis), pp. 104–13, Nov. 1992.

[13] H. Li, S. Tandri, M. Stumm, and K. Sevcik, "Locality and Loop Scheduling on NUMA Multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing*, pp. II 140 – II 127, Aug. 1993.

[14] R. H. Thomas and W. Crowther, "The Uniform System: An approach to runtime support for large scale shared memory parallel processors.," in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 245–254, Aug. 1988.

[15] R. Butler, J. Boyle, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors*. New York: Holt, Rinehart and Winston, Inc, 1987.

[16] S. F. Hummel and E. Schonberg, "Low-Overhead Scheduling of Nested Parallelism," *IBM Journal of Research and Development*, pp. 743–65, Nov. 1991.

[17] S. Madala and J. B. Sinclair, "Performance of Synchronous Parallel Algorithms with Regular Structures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 105–116, Jan. 1991.

[18] H. David, *Order Statistics*. New York: John Wiley, 1981.

[19] T. Montaut, "Ordonnancement dynamique d'un programme décomposé en sous-tâches indépendantes. analyse des performances," Master's thesis, Institut de Recherche en Informatique et Systèmes Aléatoires, Sept. 1991.

[20] M. Dayde, 1991. Private communication.

[21] F. Bodin, D. Windheiser, W. Jalby, D. Ataputta, M. Lee, and D. Gannon, "Performance Evaluation and Prediction for Parallel Algorithms on BBN GP1000," in *Proceedings of the 1990 International Conference on Supercomputing*, pp. 403–13, 1990.

[22] L. Kervella, "Etude expérimentale de la durée des tâches parallèles avec un ordonnancement dynamique," Master's thesis, Institut de Recherche en Informatique et Systèmes Aléatoires, Sept. 1991.

[23] L. S. et K.O Bowman, *Maximum Likelihood Estimation in Small Samples*. Charles Griffin and Company, 1977.

[24] V. Sarkar, "Determining Average Program Execution Times and their Variance," in *Proceedings of SIGPLAN 1989 Conference on Programming Language Design and Implementation*, vol. 34, pp. 723–732, 1969.

[25] T. Fahringer and H. Zima, "A Static Parameter-Based Performance Tool for Parallel Programs," in *Proceedings of the 1993 International Conference on Supercomputing*, (Tokyo, Japan), pp. 207–219, Association for Computing Machinery, 1993.

[26] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Experimentally Characterizing the Behavior of Multiprocessor Memory Systems: A Case Study," *IEEE Transactions on Software Engineering*, vol. 16, pp. 216–223, Feb. 1990.

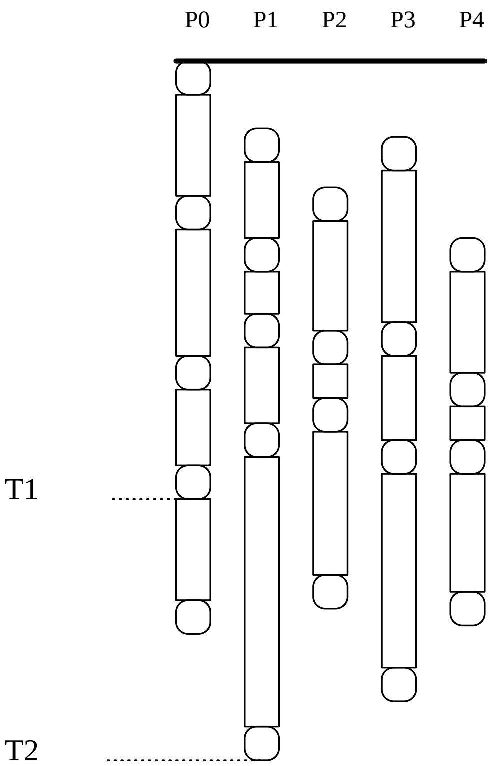

Figure 1: Program split in two phases.

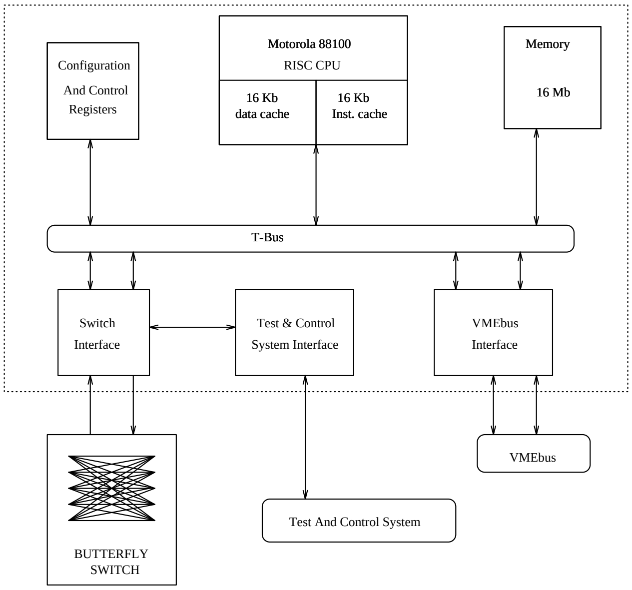

Figure 2: TC2000 processing board schematic

Memory Storage Strategy	hotspot (**H**), interleaved (**I**)
Network contention	Global/local accesses
low (**l**)	(gmax, lmax) = (80,650)
medium (**m**)	(gmax, lmax) = (160,325)
high (**h**)	(gmax, lmax) = (320,162)
Number of processors	p = **10**, **20**, **30**, **40**, **50**
Overhead	small (**s**), large (**b**)

Table 1: Experimental Parameters

```
Task(x)
int x ;
 {
  ELOG_LOG (START_TASK,x) ;
  /* Global memory accesses */
  for(i=0 ;i<gmax ;i++)
     G[i][x]=B[i][x]  ;
  ELOG_LOG (END_GLOBAL,x) ;
  /* Local calculations */
  for (i=0 ;i<lmax ;i++)
     { a=a*a+1 ;
     a=a/a+1 ;
     }
  ELOG_LOG (END_TASK,x) ;
 }
```

Figure 3: Source code of the basic task

Small Overhead								Large Overhead							
Hotspot				Interleaving				Hotspot				Interleaving			
	gmax	lmax	p		gmax	lmax	p		gmax	lmax	p		gmax	lmax	p
Hl10.s	80	650	10	*Il10.s*	80	650	10	*Hl10.b*	80	650	10	*Il10.b*	80	650	10
Hl20.s	80	650	20	*Il20.s*	80	650	20	*Hl20.b*	80	650	20	*Il20.b*	80	650	20
Hl30.s	80	650	30	*Il30.s*	80	650	30	*Hl30.b*	80	650	30	*Il30.b*	80	650	30
Hl40.s	80	650	40	*Il40.s*	80	650	40	*Hl40.b*	80	650	40	*Il40.b*	80	650	40
Hl50.s	80	650	50	*Il50.s*	80	650	50	*Hl50.b*	80	650	50	*Il50.b*	80	650	50
Hm10.s	160	325	10	*Im10.s*	160	325	10	*Hm10.b*	160	325	10	*Im10.b*	160	325	10
Hm20.s	160	325	20	*Im20.s*	160	325	20	*Hm20.b*	160	325	20	*Im20.b*	160	325	20
Hm30.s	160	325	30	*Im30.s*	160	325	30	*Hm30.b*	160	325	30	*Im30.b*	160	325	30
Hm40.s	160	325	40	*Im40.s*	160	325	40	*Hm40.b*	160	325	40	*Im40.b*	160	325	40
Hm50.s	160	325	50	*Im50.s*	160	325	50	*Hm50.b*	160	325	50	*Im50.b*	160	325	50
Hh10.s	320	162	10	*Ih10.s*	320	162	10	*Hh10.b*	320	162	10	*Ih10.b*	320	162	10
Hh20.s	320	162	20	*Ih20.s*	320	162	20	*Hh20.b*	320	162	20	*Ih20.b*	320	162	20
Hh30.s	320	162	30	*Ih30.s*	320	162	30	*Hh30.b*	320	162	30	*Ih30.b*	320	162	30
Hh40.s	320	162	40	*Ih40.s*	320	162	40	*Hh40.b*	320	162	40	*Ih40.b*	320	162	40
Hh50.s	320	162	50	*Ih50.s*	320	162	50	*Hh50.b*	320	162	50	*Ih50.b*	320	162	50

Table 2: Index to Experiments Performed

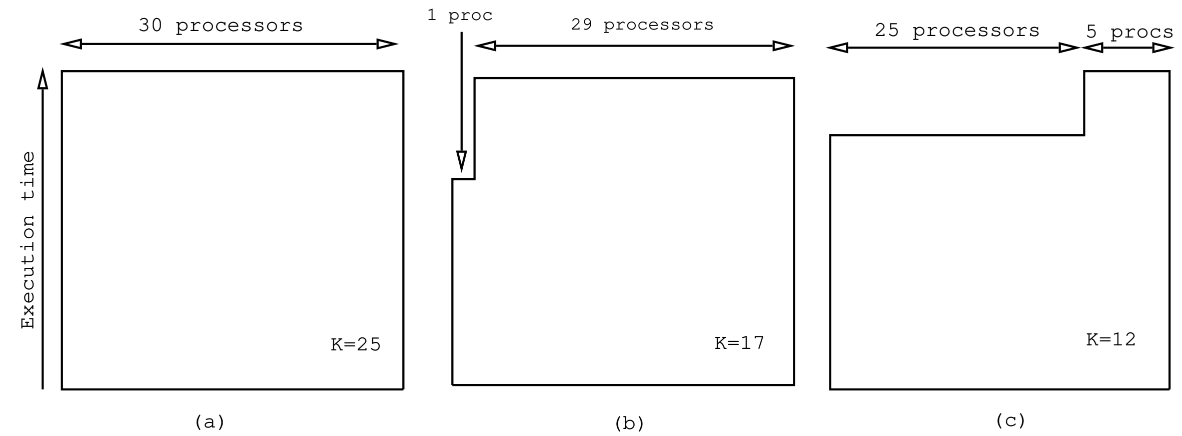

Figure 12: Impact of Choice of K on Load Imbalance

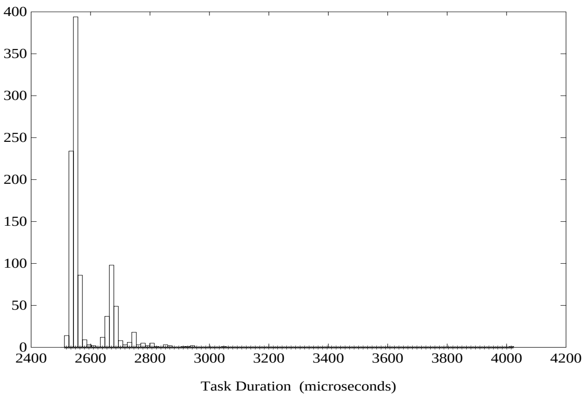

Figure 4: Task duration histogram: Experiment Il20

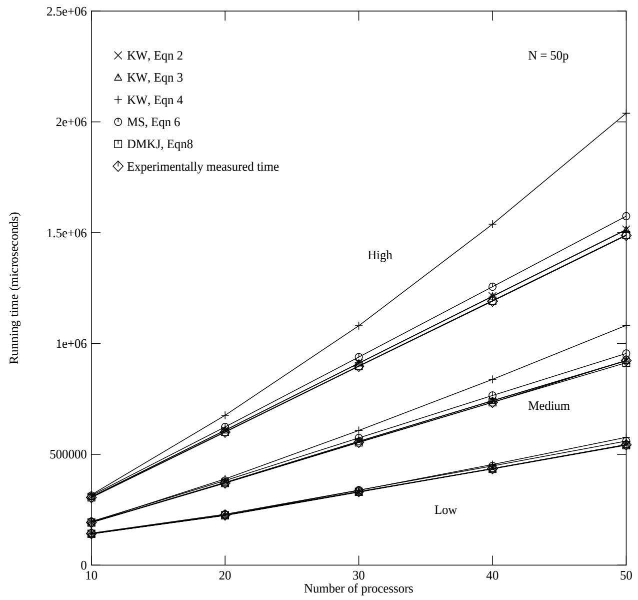

Figure 6: Predicted and actual running time for hotspot data with large overhead

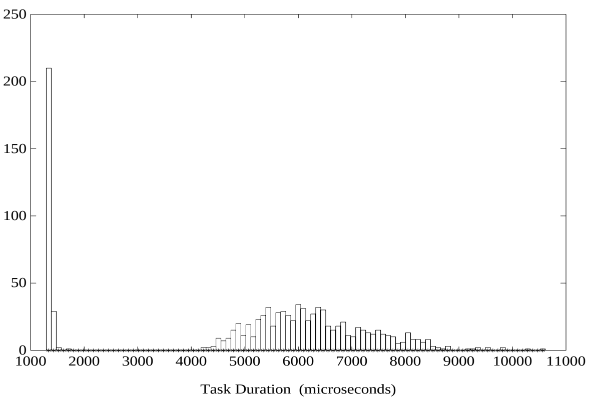

Figure 5: Task duration histogram: Experiment Hm20

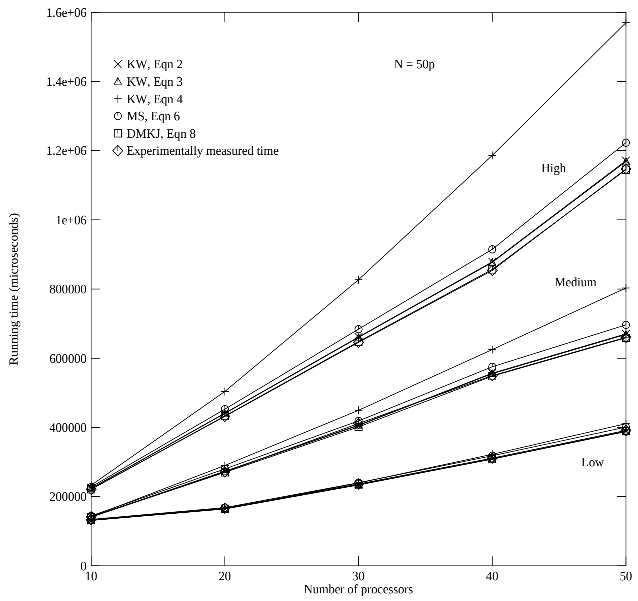

Figure 7: Predicted and actual running time for hotspot data with small overhead

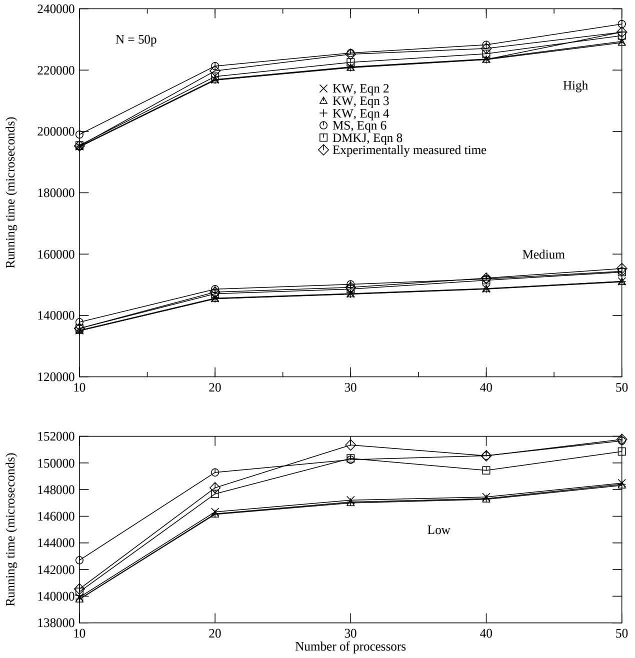

Figure 8: Predicted and actual running time for interleaved data with large overhead

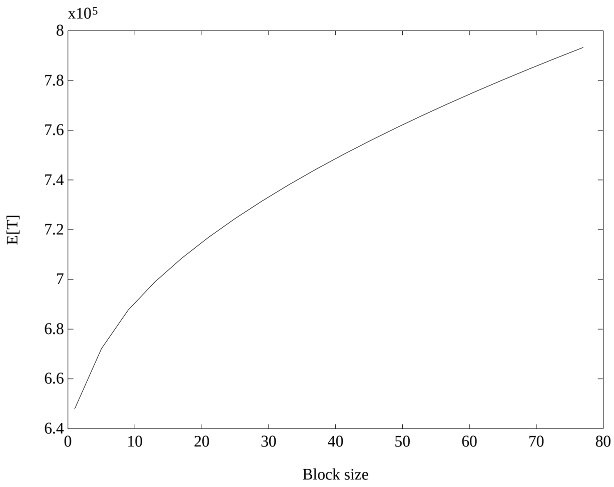

Figure 10: A Typical E[T] vs K Curve for Hotspot Experiments with Small Overheads

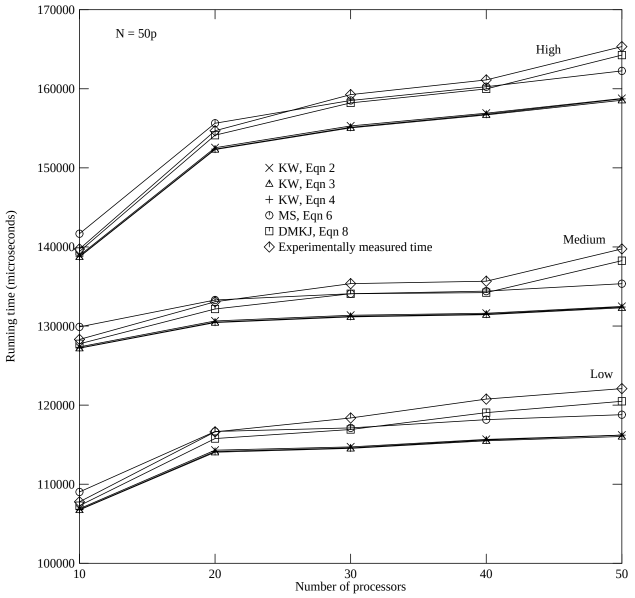

Figure 9: Predicted and actual running time for interleaved data with small overhead

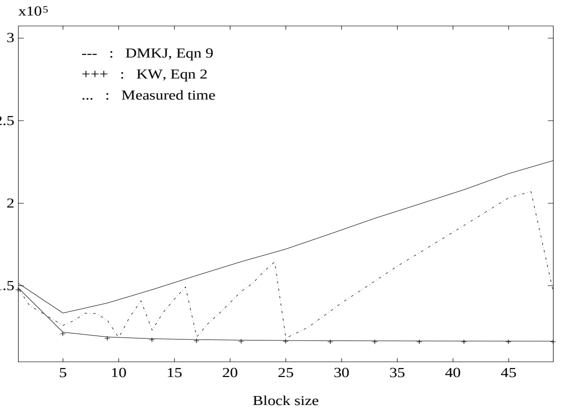

Figure 11: Total execution time (microsecs)

Hotspot

	μ	σ	μ_l	μ_g	% g	h	$E[t_i]$
Hl10.b	2509	262	1946	562	22	307	451
Hl20.b	3570	727	1947	1622	45	890	1830
Hl30.b	4920	1334	1946	2973	60	1616	3747
Hl40.b	6260	1905	1949	4310	69	2355	3021
Hl50.b	7663	2466	1946	5716	75	3078	3796
Hm10.b	2813	918	975	1838	65	1010	427
Hm20.b	5099	2344	976	4121	81	2246	1993
Hm30.b	7493	3835	978	6514	87	3481	3093
Hm40.b	9860	5204	978	8881	90	4706	3664
Hm50.b	12170	6561	1010	11686	92	5929	5007
Hh10.b	4116	2421	490	3625	88	1946	5436
Hh20.b	7944	5675	491	7453	94	3953	2384
Hh30.b	11810	8677	490	11318	96	5968	3260
Hh40.b	15680	11830	506	15169	97	7952	3091
Hh50.b	19480	14850	543	18937	98	9974	5765

Interleaving

	μ	σ	μ_l	μ_g	% g	h	$E[t_i]$
Il10.b	2496	70	1950	545	22	299	442
Il20.b	2584	75	1951	632	24	340	1387
Il30.b	2592	89	1953	638	25	347	3160
Il40.b	2599	96	1957	641	25	344	1997
Il50.b	2610	102	1961	647	25	354	2370
Im10.b	2112	69	979	1132	54	588	676
Im20.b	2247	75	980	1266	56	662	1494
Im30.b	2265	82	980	1284	57	673	1512
Im40.b	2283	103	983	1300	57	686	2754
Im50.b	2305	128	984	1320	57	710	3080
Ih10.b	2748	70	490	2368	83	1151	387
Ih20.b	3027	82	491	2536	84	1307	1130
Ih30.b	3077	110	492	2584	84	1338	2651
Ih40.b	3113	102	493	2619	84	1354	1713
Ih50.b	3167	538	502	2664	84	1390	2982

Table 3: Task Duration Statistics (microseconds)

Hotspot

	μ	σ	μ_l	μ_g	% g	h	$E[t_i]$
Hl10.s	2600	389	615	1984	0.23	17	730
Hl20.s	3250	572	1298	1951	0.38	17	1638
Hl30.s	4611	1113	2661	1950	0.54	17	3369
Hl40.s	6082	1624	4131	1950	0.64	17	3260
Hl50.s	7648	2052	5678	1969	0.71	17	5496
Hm10.s	2782	814	1803	978	0.60	16	575
Hm20.s	5333	2450	4339	993	0.72	17	2246
Hm30.s	7924	3724	6939	984	0.77	17	2845
Hm40.s	10865	4688	9859	1005	0.81	17	2260
Hm50.s	13042	6256	12056	998	0.80	17	3316
Hh10.s	4358	2476	3868	490	0.80	16	533
Hh20.s	8529	5713	8038	491	0.82	17	1623
Hh30.s	12743	8958	12252	491	0.83	17	3460
Hh40.s	16902	11990	16411	490	0.84	17	2118
Hh50.s	22615	14080	22098	517	0.86	18	4148

Interleaving

	μ	σ	μ_l	μ_g	% g	h	$E[t_i]$
Il10.s	2523	76	566	1956	0.22	20	440
Il20.s	2587	84	631	1955	0.24	21	1591
Il30.s	2596	90	636	1959	0.25	26	2821
Il40.s	2602	104	640	1962	0.25	23	2742
Il50.s	2610	111	641	1968	0.25	33	5931
Im10.s	2113	71	1134	979	0.54	21	428
Im20.s	2256	165	1261	981	0.56	21	1694
Im30.s	2266	102	1285	982	0.57	22	2308
Im40.s	2279	117	1295	984	0.57	27	3532
Im50.s	2293	127	1308	986	0.57	23	4444
Ih10.s	2754	81	2260	493	0.82	20	471
Ih20.s	3024	93	2530	493	0.84	22	1540
Ih30.s	3078	109	2585	493	0.84	22	2847
Ih40.s	3106	132	2612	494	0.84	24	2978
Ih50.s	3144	153	2648	496	0.84	22	5365

Table 4: Task Duration Statistics continued (microseconds)

Small Overheads

Hotspot						*Interleaving*					
	K=1	$K = K_{opt}^{KW}$		$K = K_{opt}^{true}$			K=1	$K = K_{opt}^{KW}$		$K = K_{opt}^{true}$	
	T	K	T	K	T		T	K	T	K	T
Hl10.s	1.331	1	1.331	NA	NA	*Il10.s*	1.283	5	1.270	10	1.270
Hl20.s	1.666	1	1.666	NA	NA	*Il20.s*	1.331	5	1.317	10	1.314
Hl30.s	2.365	1	2.365	NA	NA	*Il30.s*	1.354	5	1.330	5	1.330
Hl40.s	3.113	1	3.113	NA	NA	*Il40.s*	1.357	4	1.406	2	1.373
Hl50.s	3.915	1	3.915	NA	NA	*Il50.s*	1.398	5	1.356	5	1.356
Hm10.s	1.417	1	1.417	NA	NA	*Im10.s*	1.078	6	1.137	10	1.060
Hm20.s	2.715	1	2.715	NA	NA	*Im20.s*	1.166	3	1.168	5	1.136
Hm30.s	4.113	1	4.113	NA	NA	*Im30.s*	1.184	4	1.194	5	1.161
Hm40.s	5.498	1	5.498	NA	NA	*Im40.s*	1.208	4	1.290	10	1.186
Hm50.s	6.605	1	6.605	NA	NA	*Im50.s*	1.221	4	1.216	2	1.194
Hh10.s	2.212	1	2.212	NA	NA	*Ih10.s*	1.397	5	1.379	5	1.379
Hh20.s	4.319	1	4.319	NA	NA	*Ih20.s*	1.547	5	1.536	10	1.529
Hh30.s	6.459	1	6.459	NA	NA	*Ih30.s*	1.593	4	1.607	5	1.564
Hh40.s	8.539	1	8.539	NA	NA	*Ih40.s*	1.611	4	1.693	2	1.608
Hh50.s	1.147	1	1.147	NA	NA	*Ih50.s*	1.653	3	1.631	10	1.617

Large Overheads

Hotspot						*Interleaving*					
	K=1	$K = K_{opt}^{KW}$		$K = K_{opt}^{true}$			K=1	$K = K_{opt}^{KW}$		$K = K_{opt}^{true}$	
	T	K	T	K	T		T	K	T	K	T
Hl10.b	1.423	14	1.460	7	1.316	*Il10.b*	1.405	34	1.721	50	1.268
Hl20.b	2.259	14	1.830	17	1.747	*Il20.b*	1.481	32	1.679	25	1.318
Hl30.b	3.320	13	2.508	16	2.477	*Il30.b*	1.513	28	1.489	50	1.339
Hl40.b	4.356	13	3.326	19	3.286	*Il40.b*	1.505	26	1.423	10	1.369
Hl50.b	5.429	13	4.082	18	4.023	*Il50.b*	1.518	25	1.369	25	1.369
Hm10.b	1.924	14	1.599	11	1.487	*Im10.b*	1.358	54	1.427	50	1.067
Hm20.b	3.705	12	2.918	17	2.834	*Im20.b*	1.476	51	1.165	50	1.147
Hm30.b	5.541	11	4.754	19	4.221	*Im30.b*	1.492	46	2.077	50	1.172
Hm40.b	7.347	10	5.675	13	5.583	*Im40.b*	1.522	39	1.803	50	1.197
Hm50.b	9.231	10	6.998	13	6.941	*Im50.b*	1.553	34	1.598	50	1.226
Hh10.b	3.051	11	2.541	18	2.340	*Ih10.b*	1.953	84	2.273	50	1.399
Hh20.b	6.000	9	4.734	17	4.487	*Ih20.b*	2.197	75	2.260	50	1.541
Hh30.b	8.970	9	7.017	17	6.792	*Ih30.b*	2.252	60	1.904	50	1.585
Hh40.b	11.91	9	9.386	11	9.004	*Ih40.b*	2.270	62	1.978	25	1.638
Hh50.b	14.87	8	11.56	17	11.30	*Ih50.b*	2.324	20	1.965	50	1.661

Table 5: Running times for block scheduling (10^5 microsecs).