# Scheduling of Wavefront Parallelism on Scalable Shared-memory Multiprocessors*

Naraig Manjikian and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
email: {nmanjiki,tsa}@eecg.toronto.edu

Abstract—*Tiling exploits temporal reuse carried by an outer loop of a loop nest to enhance cache locality. Loop skewing is typically required to make tiling legal. This restricts parallelism to wavefronts in the tiled iteration space. For a small number of processors, wavefront parallelism can be efficiently exploited using dynamic self-scheduling with a large tile size. Such a strategy enhances* intratile *locality, but does not necessarily enhance* intertile *locality. We show that dynamic self-scheduling performs poorly on scalable shared-memory multiprocessors where smaller tiles are necessary to provide sufficient parallelism—smaller tiles place greater importance on intertile locality. We propose static scheduling strategies which enhance intertile locality for small tiles. Results of experiments on a Convex SPP1000 multiprocessor demonstrate that our strategies outperform dynamic self-scheduling by a factor of up to 2.3 on 30 processors.*

## 1 Introduction

Scalable shared-memory multiprocessors (SSMMs) have become increasingly viable as platforms for high-performance computing by efficiently supporting coherent shared memory in hardware for large numbers of processors [2]. Examples include the Convex SPP1000 [3], the Stanford FLASH [5], and the University of Toronto NUMAchine [12]. Although the memory in SSMMs is logically shared by all processors, it is physically distributed to provide scalability, as shown in Figure 1. As a result, memory accesses are non-uniform; access latency for remote memory is considerably higher than for local memory. SSMMs heavily rely on data caching to reduce the effective memory access latency. Consequently, both enhancing locality and exploiting parallelism are instrumental in achieving scaling performance.

A common locality-enhancing technique for loop nests is *tiling*, also known as *blocking* [1]. Tiling is particularly effective in exploiting temporal reuse carried by an outer loop. Iterations from the original loop nest are reordered and blocked into units of *tiles*. The data used by one tile is loaded once into the cache and retained there for subsequent reuse during the execution of the tile. A loop nest can be legally tiled if it is fully permutable, i.e., if none of its loop-carried dependence vectors have negative elements [13]. Loop skewing [1] is used to eliminate any negative elements and produce a fully-permutable loop nest. However, loop skewing also results in loop-carried dependences in outer loops after tiling. These loop-carried dependences limit the available coarse-grain parallelism to *wavefronts* in the tiled iteration space. Tiles within each wavefront may be executed concurrently, but synchronization between wavefronts is required to satisfy dependences.

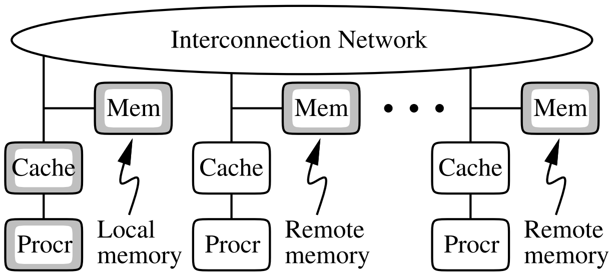



Figure 1: Scalable shared-memory multiprocessor

In this paper, we consider the problem of *scheduling* the execution of a tiled loop nest to exploit the wavefront parallelism on scalable shared-memory multiprocessors. We demonstrate that the commonly-used strategy of dynamic self-scheduling, while adequate for small-scale multiprocessors, performs poorly on SSMMs. This is because dynamic self-scheduling cannot simultaneously enhance locality and provide sufficient parallelism. We apply two static scheduling strategies in a manner which enhances locality for a large number of processors while still providing sufficient parallelism. We provide analysis and experimental results on a Convex SPP1000 multiprocessor to demonstrate the superiority of our static scheduling strategies.

The remainder of this paper is organized as follows. Section 2 gives background on tiling loop nests and wavefront parallelism. Section 3 presents the two static scheduling strategies and analytically evaluates their benefits. Section 4 provides a comparative experimental evaluation of all three scheduling strategies on the Convex. Section 5 briefly outlines related work. Finally, Section 6 gives concluding remarks.

*This research is supported by grants from NSERC (Canada) and ITRC (Ontario). The use of the Convex SPP1000 was provided by the University of Michigan Center for Parallel Computing.

```
do t=1,T
  do j=2,N−1
    do i=2,N−1
      a[i,j] = (a[i,j]+a[i+1,j]+a[i−1,j]
          +a[i,j+1]+a[i,j−1]) / 5
```

(a)

```
do jj=2,N−1+T,B
  do ii=2,N−1+T,B
    do t=1,T
      do j=max(jj,2+t),min(jj+B−1,N−1+t)
        do i=max(ii,2+t),min(ii+B−1,N−1+t)
          a[i−t,j−t] = (a[i−t,j−t]+a[i+1−t,j−t]+a[i−1−t,j−t]
              +a[i−t,j+1−t]+a[i−t,j−1−t]) / 5
```

(b)

Figure 2: Tiling the SOR loop nest

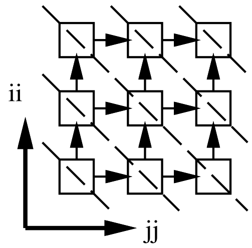



Figure 3: Dependences and wavefronts

## 2 Tiling and Wavefront Parallelism

The SOR loop nest in Figure 2(a) is used to illustrate tiling and the resulting wavefront parallelism. The outer loop carries reuse and the dependences for this loop nest have the distance vectors: $\{(1,0,0), (1,-1,0), (1,0,-1), (0,1,0), (0,0,1)\}$. The loop nest is not fully permutable, and loop skewing must be applied. Both inner loops $i$ and $j$ are skewed by one iteration with respect to loop $t$, resulting in the transformed distance vectors: $\{(1,1,1), (1,0,1), (1,1,0), (0,1,0), (0,0,1)\}$. The loop nest can be then tiled legally by first strip-mining the skewed $i$ and $j$ loops by a factor of $B$, then by permuting the resulting $ii$ and $jj$ loops to the outermost level, as in Figure 2(b). The $ii$ and $jj$ loops carry the same dependences as the $i$ and $j$ loops prior to tiling, hence neither of the outer loops can be executed in parallel. The only available parallelism is along the wavefronts shown in Figure 3; each square represents a $B \times B$ tile of iterations from the original loop nest.

There are two general approaches for exploiting wavefront parallelism. The first is to use additional loop skewing to obtain DOALL loops which reflect the parallelism in each wavefront. For the SOR example, skewing the tiled iteration space shown in Figure 3 yields the iteration space shown in Figure 4. The $ii$ loop iterations may be executed in parallel. The outermost $jj$ loop remains sequential, requiring global synchronization of all processors between successive iterations. Because the number of tiles that can be executed in parallel varies in each wavefront, processors may not be fully utilized between global synchronizations. For example, the middle wavefront in Figure 4 has three tiles labelled X, Y, and Z. With two processors executing in parallel, both processors are initially busy executing tiles X and Y. However, one processor must remain idle until the remaining tile Z is executed because of the global synchronization required for the DOALL loop.

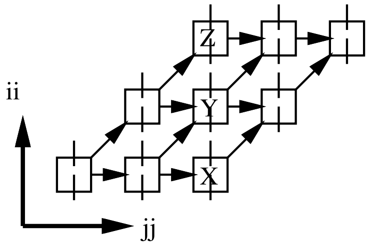


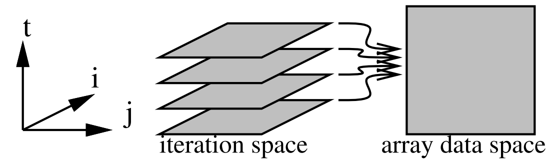

Figure 4: Exploiting parallelism with inner DOALL loops

The second approach to exploit wavefront parallelism is to treat the two outer loops as DOACROSS loops and execute the independent tiles in each wavefront with dynamic self-scheduling. Processors which become idle obtain the next unassigned tile. Loop-carried dependences are enforced by appropriate synchronization prior to executing the tile to ensure that adjacent tiles in the preceding wavefront have been completed. This approach avoids global synchronization and effectively utilizes idle processors, allowing portions of different wavefronts to proceed concurrently, albeit local synchronization is required between tiles. Consequently, dynamic self-scheduling has been the approach of choice in scheduling wavefront parallelism [13].

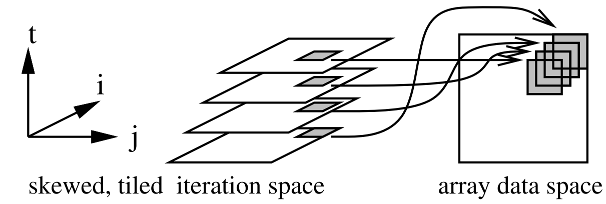
### 2.1 Tile Size, Parallelism, and Locality

The tile size has a significant impact on the performance of a tiled loop nest because it determines both the degree of parallelism and the extent to which locality is enhanced. With wavefront parallelism, a smaller tile size increases the number of wavefronts and, more importantly, increases the number of independent tiles in each wavefront. Hence, the degree of parallelism increases with smaller tile sizes, although the frequency of synchronization also increases.

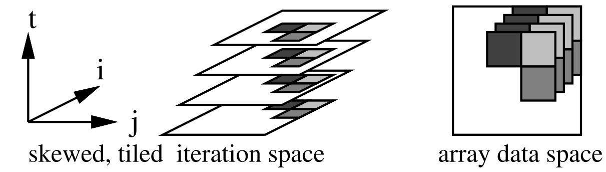
The tile size also dictates the extent of locality enhancement when loop skewing is required for tiling. We classify locality into two types: *intratile* locality and *intertile* locality. Intratile locality is the primary result of applying tiling to capture the temporal reuse carried by the outer loop in the original loop nest; data for each tile is loaded once into the cache, then reused within the same tile. Intertile locality results from exploiting temporal reuse across adjacent tiles. Such reuse arises from the data access patterns created by loop skewing, as illustrated for SOR in Figure 5. The iteration and data spaces of the original SOR loop nest are shown in Figure 5(a). With loop skewing and tiling, each iteration of the original outer loop executed within a tile accesses a slightly different portion of the array, as shown in Figure 5(b). Adjacent tiles in the iteration space access overlapping regions in the data space, as shown




(a) Iteration and data spaces for original SOR loop nest




(b) Skewed data access patterns within each tile




(c) Overlapping array regions swept by adjacent tiles

Figure 5: Intertile data reuse for the SOR loop nest

in Figure 5(c). This overlap gives rise to intertile reuse. When adjacent tiles are executed by the same processor, data elements in overlapping regions are retained in the cache, which reduces the number of cache misses incurred to load the data referenced in successive tiles. That is, data in the overlapping regions is loaded only once then reused not only within a tile, but also in adjacent tiles. In this case, intertile locality is enhanced. On the other hand, when adjacent tiles are executed by different processors, cache misses are incurred by each processor to load all the data referenced within each tile, including the data in the overlapping regions. In this case, intertile locality is lost.

The impact of tile size on intratile and intertile locality is illustrated in Figure 6. The shaded regions represent the data accessed by adjacent tiles, as in Figure 5(c). The overlapping regions correspond to the intersection of the data accessed by different tiles. For a given number of iterations in the original outer loop, the amount of data in the overlapping regions is relatively small compared to the total amount of data accessed by the tile when the tile size is large. Consequently, a large tile size enhances intratile locality and diminishes the impact of intertile locality. In contrast, for the same number of iterations and a small tile size, the amount of data in the overlapping regions is a much larger fraction of the total amount of data accessed by the tile. Hence, a small tile size increases the importance of intertile locality.

## 2.2 Self-scheduling and Intertile Locality

Dynamic self-scheduling is adequate for exploiting wavefront parallelism in tiled loop nests for small-scale shared-memory multiprocessors. With a limited number

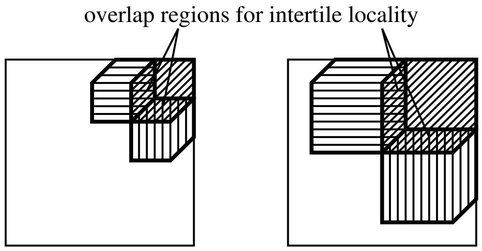



Figure 6: Impact of tile size on locality

of processors, a large tile size generally provides an adequate degree of parallelism. Consequently, intratile locality is enhanced because a large tile size captures most of the reuse from the original loop nest within a single tile, and intertile locality has little impact on performance.

However, dynamic self-scheduling is not appropriate for scalable shared-memory multiprocessors for two reasons. First, a large number of processors requires a relatively smaller tile size to result in sufficient parallelism. A small tile size reduces intratile locality and places greater importance on intertile locality. Dynamic self-scheduling is not likely to enhance intertile locality since tiles are assigned arbitrarily to idle processors. The second reason is that cache misses which result from the reduced intertile locality with small tile sizes are likely to be incurred for remote, rather than local, memory due to the arbitrary assignment of tiles to processors. The performance degradation resulting from these misses may be significant.

# 3 Scheduling for Intertile Locality

In this section, we describe how two common scheduling strategies, namely *static cyclic* and *static block*, can be applied to exploit wavefront parallelism in a manner which enhances intertile locality for the smaller tile sizes which are necessary to provide sufficient parallelism on SSMMs. The static scheduling strategies are then compared with dynamic self-scheduling on the basis of runtime overhead, synchronization requirements, and locality enhancement.

## 3.1 Static Cyclic Scheduling

Static cyclic scheduling assigns rows of horizontally-adjacent tiles to the same processor, as shown in Figure 7. In this manner, temporal intertile reuse along each row of tiles is exploited by one processor to enhance intertile locality. The cyclic mapping of rows of tiles to processors distributes the workload in each wavefront evenly among processors to fully exploit the available parallelism.

Static cyclic scheduling improves over dynamic self-scheduling in three ways. First, cyclic scheduling enhances intertile locality for horizontally-adjacent tiles, whereas dynamic self-scheduling does not necessarily exploit any intertile reuse. Second, synchronization to enforce loop-carried dependences is required only for vertically-adjacent tiles, since horizontally-adjacent tiles are executed in the correct order by the same processor. Third, the scheduling

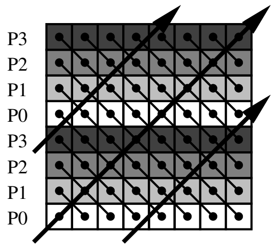



Figure 7: Static cyclic scheduling of tiles

overhead is reduced since the assignment of tiles to processors is determined statically. However, cyclic scheduling still requires synchronization for each tile to enforce dependences, and not all of the intertile reuse is exploited.

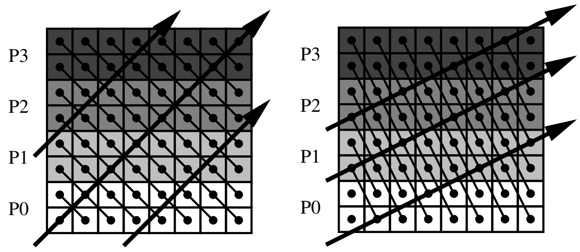
## 3.2 Static Block Scheduling

Static block scheduling assigns contiguous blocks of tiles to the same processor, as shown in Figure 8. In this manner, all of the intertile reuse within a block of horizontally- and vertically-adjacent tiles is exploited by one processor to enhance intertile locality. However, the parallelism is not exploited efficiently for the original wavefronts shown in Figure 8(a) because a portion of the processors is left idle for the few initial and few final wavefronts. The block assignment of tiles to processors precludes the use of additional processors even when there is a sufficient number of tiles which can be executed in parallel. Consequently, it takes longer for all processors to become active, and it takes longer for execution to complete.

Block scheduling requires the use of modified wavefronts as shown in Figure 8(b) to provide greater parallelism. This involves rotating wavefronts such that the number of independent tiles in the largest wavefront is exactly equal to the number of processors. This rotation corresponds to the selection of a different *scheduling vector*[1] [4]. The scheduling vector is $(1, 1)$ for the original wavefronts in Figure 8(a). The scheduling vector for the modified wavefronts in Figure 8(b) is given by $(\lfloor(N + T)/(B \cdot P)\rfloor, 1)$, where $N + T$ is the number of iterations (with skewing), $B$ is the tile size, and $P$ is the number of processors. The new scheduling vector preserves the loop-carried dependences, but reduces the time before all processors become active in parallel execution and reduces the completion time.

Static block scheduling improves over both dynamic and cyclic scheduling in two ways. First, block scheduling exploits all intertile reuse, except at block boundaries. Second, synchronization to enforce loop-carried dependences is required only for tiles on block boundaries; no synchronization is required for adjacent interior tiles, since they are executed in the correct order by the same processor. Similar to cyclic scheduling, the scheduling overhead is also reduced since the assignment of tiles to processors is determined statically.

[1]A scheduling vector is orthogonal to the wavefronts, and hence defines their orientation in the tiled iteration space.




(a) Original wavefronts (b) Modified wavefronts

Figure 8: Static block scheduling of tiles

## 3.3 Comparison of Scheduling Strategies

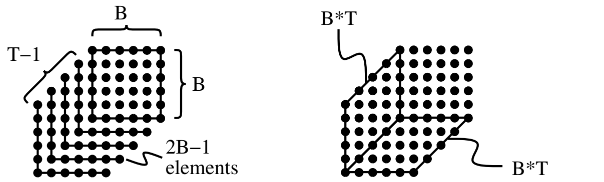
The scheduling strategies are compared on the basis of *runtime overhead*, *synchronization*, *parallelism*, and *intertile locality enhancement*. The features of each strategy are summarized in Table 1. Dynamic self-scheduling incurs runtime overhead, and requires synchronization for both horizontally- and vertically-adjacent tiles. A synchronization counter is needed for each row of tiles. Static cyclic scheduling requires synchronization only for vertically-adjacent tiles, but counters are still required for each row of tiles. Finally, static block scheduling requires synchronization only for vertically-adjacent tiles on block boundaries. The number of synchronization counters required is therefore equal only to the number of processors.

The impact of the reduced parallelism for block scheduling can be shown by determining the theoretical completion time for each strategy, which is given in Table 1. We define a time step as the theoretical execution time for each tile (i.e., neglecting the impact of synchronization and locality); the completion time is expressed in these units. For simplicity, we assume no variance in the computation per tile. For the example in Figure 8(b), block scheduling takes 22 time steps to execute all tiles. In contrast, dynamic and cyclic scheduling with the same tile size take only 19 time steps. The reduction in parallelism for block scheduling can be mitigated with a smaller tile size that allows the use of a scheduling vector which further reduces the time before all processors become active.

The extent of intertile locality enhancement for each scheduling strategy is shown in Table 1. The importance of enhancing intertile locality can be demonstrated by estimating the total latency for cache hits and misses that occur during the execution of a single tile. Again, we use the SOR loop nest as an example. For a tile size of $B \times B$, and $T$ iterations in the original outer loop of the loop nest being tiled, the total number of data accesses to the cache within each tile is given by $B^2T$. This number is conservative because register locality reduces the number of cache accesses. Each access to the cache has a latency of $C$ clock cycles. Some fraction of these references miss in the cache and incur the additional cache miss latency $M$. For dynamic self-scheduling, there is no intertile locality, and in the worst case, misses are incurred for all data elements accessed *for the first time* within the tile. The number of such elements is given by $B^2 + (2B - 1)(T - 1)$,

Table 1: Comparison of scheduling strategies for tiling

| | Dynamic | Cyclic | Block |
|---|---|---|---|
| runtime overhead | yes | no | no |
| synch., #counters | horizontal/vertical *tiles*, $\lceil \frac{N+T}{B} \rceil$ | vertical *tiles*, $\lceil \frac{N+T}{B} \rceil$ | vertical *processors*, $P$ |
| completion time | $\lceil \frac{N+T}{B} \rceil^2 / P + P - 1$ | $\lceil \frac{N+T}{B} \rceil^2 / P + P - 1$ | $\left(P - 1 + \lceil \frac{N+T}{B} \rceil\right) \cdot \lceil \frac{N+T}{B \cdot P} \rceil$ |
| intertile locality | none | horizontal tiles | horizontal/vertical tiles |



(a) Elements per tile (b) Elements per overlap region

Figure 9: Number of data elements within a tile

as shown in Figure 9(a). This number must then be divided by $L$, the cache line size, to arrive at an *estimate* for the number of cache misses. The total latency in clock cycles for memory accesses is then given by multiplying by the cache miss latency $M$. Finally, the total memory access latency, including the cache accesses is given by $B^2TC + (B^2 + (2B-1)(T-1))(M/L)$. To measure the extent of locality enhancement for different values of $B$ and $T$, it is useful to express the fraction $f$ of the total memory access latency per tile that is due to cache misses, which is given by

$$f_{dyn} = \frac{(B^2 + (2B-1)(T-1))(M/L)}{B^2TC + (B^2 + (2B-1)(T-1))(M/L)}.$$

A similar derivation can be made for static cyclic scheduling and static block scheduling. Because there is intertile locality for adjacent tiles, fewer misses are incurred per tile. The reduction in the number of misses is determined by the number of elements in one or both of the overlap regions shown in Figure 9(b). Once again, the fraction of the latency due to misses can be determined. Hence,

$$f_{cyc} = \frac{(B^2 + BT - 2B - T + 1)(M/L)}{B^2TC + (B^2 + BT - 2B - T + 1)(M/L)},$$

and

$$f_{blk} = \frac{(B^2 - 2B + 1)(M/L)}{B^2TC + (B^2 - 2B + 1)(M/L)}.$$

Figure 10 plots the fraction $f$ for different tile sizes $B$ and different values of $T$. The cache line size is $L = 4$ elements, the cache access latency is $C = 1$ clock cycle, and the cache miss latency is $M = 50$ clock cycles. As $T$ increases, $f$ decreases for all three strategies because reuse carried by the original outer loop is captured within

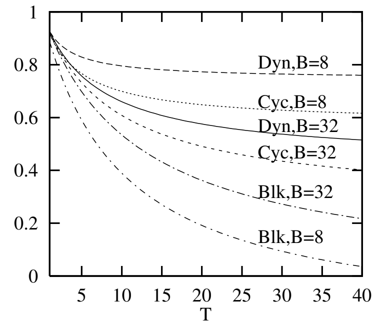



Figure 10: Fraction of miss latency per tile

the tile through intratile locality. However, $f$ decreases far more rapidly for block scheduling. This is because block scheduling benefits from enhancing *intertile* locality by reducing the number of cache misses by an amount proportional to the overlap regions in Figure 9(b). Furthermore, for a given value of $T$, $f$ is further reduced with a smaller tile size for block scheduling because intertile locality is more critical when the tile size is small (see Figure 6). In contrast, for a given value of $T$, $f$ *increases* when the tile size is reduced for both dynamic and cyclic scheduling. This is because dynamic and cyclic scheduling do not enhance intertile locality to the same extent for small tile sizes as block scheduling.

In conclusion, all the scheduling strategies considered provide sufficient parallelism with small tile sizes, but small tiles require exploiting intertile reuse for locality. Dynamic scheduling does not exploit any intertile reuse. Cyclic scheduling exploits some intertile reuse while providing the same parallelism for the same tile size. Hence, it is expected to perform better than dynamic scheduling. Block scheduling exploits all intertile reuse, but with slightly less parallelism than either dynamic or cyclic scheduling for the same tile size. However, the benefit of enhancing locality with block scheduling may outweigh the loss of some parallelism and provide the best overall performance. The relative performance of the three strategies for small tile sizes on a large number of processors depends on the tradeoff between parallelism and locality. This tradeoff is explored experimentally in the next section.

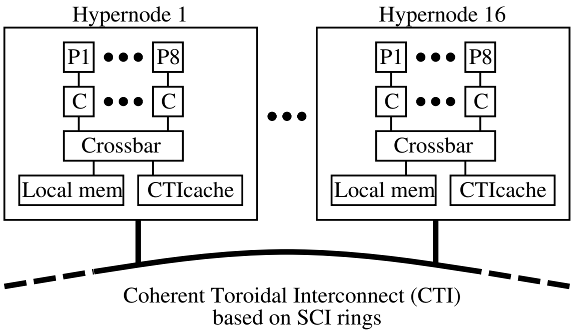
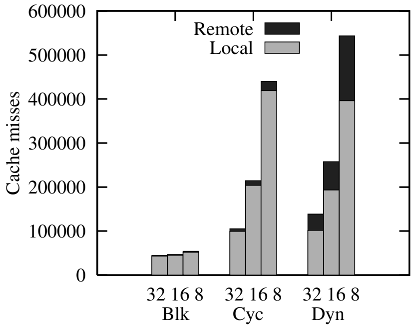



Figure 11: Architecture of the Convex SPP1000

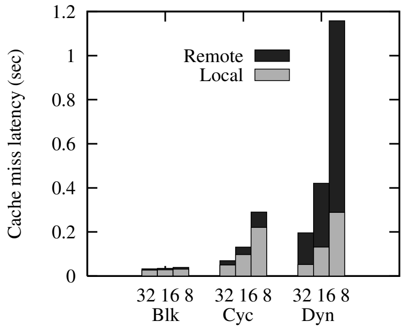
## 4 Experimental Evaluation

We report the results of experiments conducted on a Convex SPP1000 Exemplar multiprocessor [3]. The Convex SPP1000 consists of up to 16 *hypernodes*, each containing 8 processors with a crossbar connection to 512 Mbytes of common memory, as shown in Figure 11. The crossbar provides uniform access to the local memory for processors within a hypernode. Each processor is a Hewlett-Packard PA7100 RISC microprocessor running at 100 MHz with separate 1-Mbyte instruction and data caches. The cache access latency is 1 clock cycle or 10 nsec, and the cache line size is 32 bytes. Hypernodes are connected together with the Coherent Toroidal Interconnect (CTI), a system of rings based on the SCI standard interconnect, clocked at 250 MHz. The CTI permits processors to access memory in any hypernode through coherent global shared memory.

The Convex SPP1000 is a non-uniform memory access (NUMA) multiprocessor. Cache misses to retrieve data from the local hypernode memory incur a latency of 50 cycles, or 500 nsec. However, misses to retrieve data from remote hypernode memory through the CTI incur a latency of 200 cycles, or 2 $\mu$sec. A unique feature of the Convex SPP1000 is the CTIcache, which is a portion of the memory in each hypernode reserved for caching data from other hypernodes in order to reduce the effective memory latency for remote memory accesses. Remote data is retrieved in units of 64 bytes, but supplied to processors in 32-byte cache lines from the CTIcache (i.e., processors do cache remote data). The remote memory access latency is incurred once to load data into the CTIcache, and subsequent accesses by any processor which hit in the CTIcache incur the same access latency as the local memory, i.e., 50 cycles instead of 200 cycles. The Convex SPP1000 provides hardware monitoring for accurate measurement of the number of cache misses and the corresponding latencies to local and remote memory. Our experiments were conducted on a 32-processor Convex SPP1000 consisting of 4 hypernodes. The CTIcache size in each hypernode is 16 Mbytes. Two processors are reserved for system use, leaving 30 processors available for experimentation.



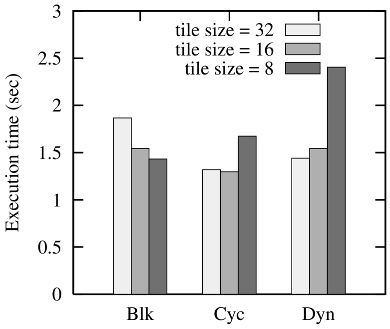

(a) Avg. cache misses for 16 processors



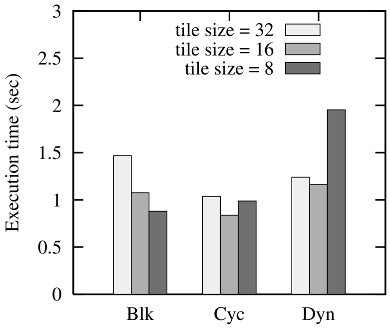

(b) Avg. miss latency for 16 processors

Figure 12: Cache misses for tiled SOR

We first report results obtained from parallel execution of the tiled SOR loop nest. The array size is $1024 \times 1024$ elements, and each element is an 8-byte floating point value. The number of iterations in the original outer loop is $T = 40$. We use tile sizes of $32 \times 32$, $16 \times 16$, and $8 \times 8$ for each of the three scheduling strategies. Larger tile sizes are not considered because they do not provide sufficient parallelism for a large number of processors. Figure 12 shows the average number of cache misses and corresponding miss latencies per processor on 16 processors (i.e., 2 hypernodes). Both the number of misses and the latencies are broken down into local and remote. Figure 12(a) indicates that block scheduling incurs far fewer misses for a given tile size than dynamic or cyclic scheduling, which agrees with the analytical observations in Section 3. The fraction of misses to remote memory is small for block and cyclic scheduling (4% and 5% respectively for a tile size of 8). This fraction is significantly larger for dynamic self-scheduling (27% for a tile size of 8). Hence, the impact of the remote misses on the total cache miss latency shown in Figure 12(b) is more pronounced for dynamic self-scheduling because of the higher cost of remote misses. As the tile size is reduced, both the number of cache misses and the total miss latencies increase dramatically for both dynamic and cyclic scheduling. In particular, the resulting miss latency for dynamic self-scheduling with a tile size of 8 is *30* times larger than for block scheduling. This clearly




(a) 16 processors




(b) 30 processors

Figure 13: Execution times for tiled SOR

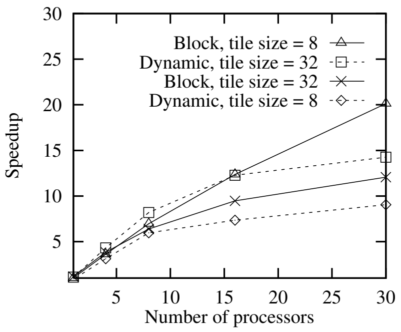



Figure 14: Speedup for tiled SOR

demonstrates the detriment of failing to provide intertile locality when the tile size is small. Block scheduling is much less sensitive to a reduction in the tile size because it exploits all intertile reuse.

The effect of the cache behavior on execution time for the tiled SOR loop is shown in Figure 13 for 16 processors, and also for 30 processors. The results indicate that static scheduling performs better than dynamic scheduling for a large number of processors, but only block scheduling improves consistently when the tile size is reduced to provide greater parallelism. Although the results indicate that cyclic scheduling with an intermediate tile size may

```
do t=1,T
  do j=2,N−1
    do i=2,N−1
      b[i,j] = (a[i+1,j]+a[i−1,j]
               +a[i,j+1]+a[i,j−1]) / 4
  do j=2,N−1
    do i=2,N−1
      a[i,j] = b[i,j]
```

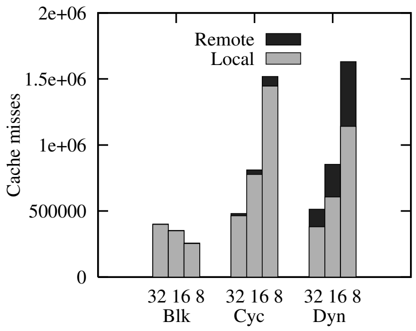
Figure 15: The Jacobi loop nest sequence

perform better than block scheduling, it may be difficult to predict an optimal tile size for cyclic scheduling which achieves the appropriate balance between sufficient parallelism and sufficient locality. Later results in this section will confirm this observation. Furthermore, block scheduling simplifies the selection of the tile size for a large number of processors. It is sufficient to choose a small tile size for greater parallelism; intertile locality is preserved with block scheduling. Hence, we focus on comparing block scheduling with dynamic scheduling for the largest and smallest tile sizes.

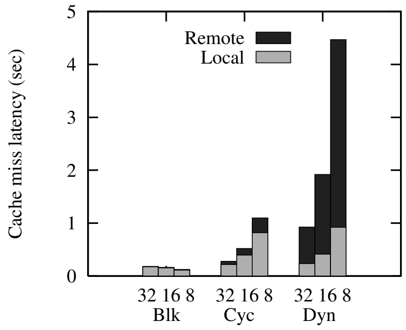
Finally, the parallel speedup of tiled SOR for various numbers of processors over the sequential *untiled* loop nest executed on a single processor is shown in Figure 14. The speedup of block scheduling and dynamic self-scheduling are compared for the largest and smallest tile sizes. When the number of processors is 8 or less, all memory accesses are confined within single hypernode, i.e., there are no remote memory accesses. Dynamic self-scheduling with a large tile size generates sufficient parallelism for the relatively small number of processors, and maximizes intratile locality. The larger tile size and the uniform memory access within a hypernode diminish the impact of intertile locality. Consequently, dynamic self-scheduling with the largest tile size performs the best. However, as the number of processors increases, a large tile size limits the speedup of dynamic self-scheduling due to insufficient parallelism. In addition, memory accesses span hypernodes and become non-uniform, which limits the speedup of dynamic self-scheduling, particularly when a smaller tile size is used to provide greater parallelism. Intertile locality is critical for small tile sizes, and dynamic self-scheduling does not exploit intertile reuse. In contrast, block scheduling with a small tile size provides sufficient parallelism while enhancing intertile locality, improving the speedup by a factor of 1.4 over dynamic self-scheduling at 30 processors.

Our scheduling techniques are applicable to any application which is tiled to exploit reuse carried by an outer loop. We present results for two more applications: the Jacobi loop nest sequence and the LL18 kernel from the Livermore Loops benchmark.

The Jacobi loop nest consists of two loop nests surrounded by an outer loop, as shown in Figure 15. There is reuse between the inner two loop nests in addition to the reuse carried by the outer loop. Tiling requires *fusion* [1] of the inner two loop nests to produce a single loop nest.

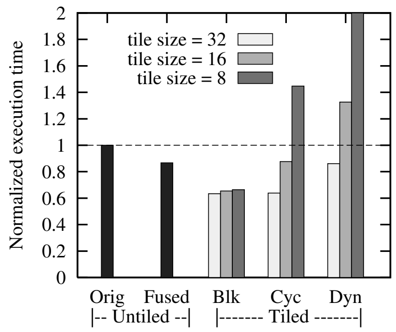



(a) Avg. cache misses for 16 processors

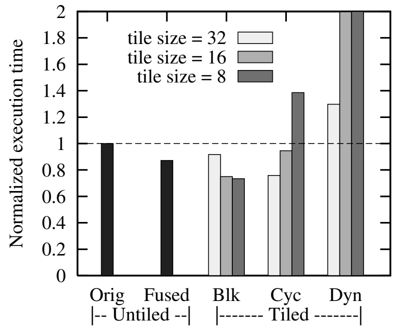



(b) Avg. miss latency for 16 processors

Figure 16: Cache misses for tiled Jacobi




(a) 16 processors




(b) 30 processors

Figure 17: Execution times for tiled Jacobi

Fusion exploits the reuse between the two inner loop nests in addition to enabling tiling. Dependences between the inner two loop nests require the application of *shift-and-peel* [9] to enable legal fusion. Once a single loop nest is obtained with fusion, loop skewing is required just as for the SOR loop nest to obtain a fully-permutable loop nest. The application of shift-and-peel to enable fusion results in dependences which require skewing the inner loops by *two* iterations with respect to the outer loop, rather than one as required for SOR. Once skewed, the loop nest is then tiled to exploit the reuse carried by the outer loop, and any of the three scheduling strategies discussed in the paper can be applied for parallel execution of the tiled loop nest. The final code is not shown due to space limitations, but is similar to the tiled SOR loop nest.

Figure 16 shows the average number of cache misses and corresponding miss latencies per processor for parallel execution of tiled Jacobi with the different scheduling strategies on 16 processors. The array sizes are $2048 \times 2048$ and the number of iterations in the original outer loop is $T = 10$. As before, the number of misses and the latencies are broken down into local and remote. The results are similar to those obtained for SOR. Block scheduling incurs the fewest cache misses as well as having the smallest fraction of remote misses. The cache latency for block scheduling is also the lowest. Dynamic self-scheduling incurs the greatest number of cache misses and a larger fraction of remote misses, which results in a dramatic increase in cache miss latency as the tile size is reduced.

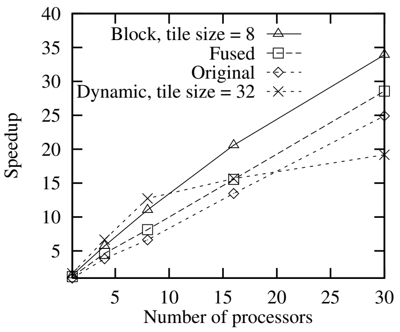
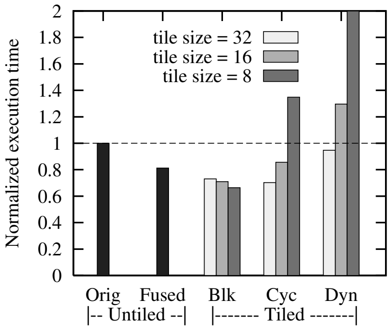
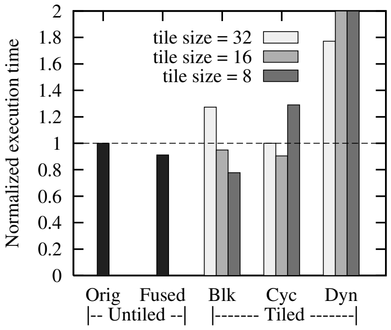



Figure 18: Speedup for Jacobi

Normalized execution times for tiled Jacobi on 16 and 30 processors are shown in Figure 13. All execution times are normalized with respect to time obtained with parallel execution of the *original* code to facilitate comparison. The normalized execution time for fusion of the inner loops without tiling is also shown, since parallel execution of the fused loops is enabled by the shift-and-peel transformation [9]. Once again, the results for tiling are similar to those obtained for SOR. The dramatic increase in execution time for dynamic self-scheduling correlates with the




(a) 16 processors




(b) 30 processors

Figure 19: Execution times for tiled LL18

increase in the cache miss latency. Block scheduling with a small tile size performs far better. Fusion exploits reuse between the inner two loops, but tiling goes further to exploit the reuse carried by the outer loop. To ensure that the full benefit of tiling is realized, the tiled loop nest must be scheduled appropriately.

Finally, the parallel speedup of tiled Jacobi for various numbers of processors over the original code executed on a single processor is shown in Figure 18. The speedup for block scheduling and dynamic self-scheduling is compared to the speedup from parallel execution of the original code and the fused version. The speedup for cyclic scheduling is not shown because its performance for small tile sizes is worse than block scheduling. Once again, dynamic self-scheduling with a large tile size is only effective in the absence of remote memory accesses, i.e., when the number of processors is 8 or less. In contrast, block scheduling with a small tile size improves the speedup by a factor of 1.8 over dynamic self-scheduling at 30 processors, and consistently outperforms even the parallel versions of the original and fused code.

The LL18 kernel from the Livermore Loop benchmark consists of *three* loop nests surrounded by an outer loop. A total of nine arrays are used, and there is reuse between the inner loop nests in addition to the reuse carried by the outer loop. Tiling requires fusion with the shift-and-peel transformation to produce a single loop nest, followed by

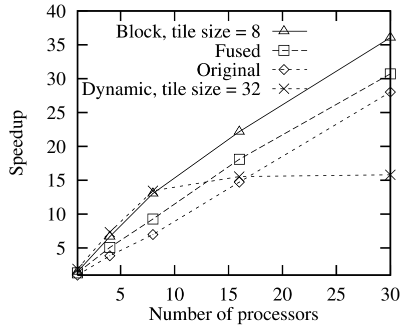



Figure 20: Speedup for LL18

skewing of the inner loops by *three* iterations. The tiled loop nest is scheduled with the different strategies just as for SOR and Jacobi. Normalized execution times for 16 and 30 processors are shown in Figure 19 for array sizes of $1024 \times 1024$ and $T = 10$ iterations in the original outer loop. The results are similar to those obtained for Jacobi. Fusion improves performance by exploiting reuse between the inner loop nests, but tiling with an appropriate scheduling strategy exploits all the reuse for the best performance. Once again, only block scheduling is successful in enhancing locality when the tile is reduced to provide sufficient parallelism for a large number of processors. The speedups for LL18 shown in Figure 20 also agree with the trends observed for Jacobi. Block scheduling improves the speedup at 30 processors by a factor of 2.3 over dynamic self-scheduling.

## 5 Related Work

There exists a large body of work dealing with the scheduling of parallel DOALL loops. Many scheduling strategies have been proposed to strike a balance between load balance and scheduling overhead. Examples include static scheduling [1], self-scheduling [1], guided self-scheduling [11], and factoring [6] to name a few. However, these strategies are not applicable when loops carry dependences, such as in wavefront parallelism.

Markatos and LeBlanc [10] propose Affinity-based Scheduling (AFS), and Li et al. [7] propose Locality-based Dynamic Scheduling (LDS) to schedule parallel DOALL loop iterations for locality as well as load balance. Both techniques address locality by initially assigning each processor a local set of independent iterations from a DOALL loop in a manner which corresponds to a distribution of the data. Load balance is promoted by having processors that become idle retrieve iterations from other processors. Neither AFS nor LDS are applicable for scheduling tiled loop nests with loop-carried dependences.

The scheduling of wavefront parallelism has been addressed by Wolf and Lam [13, 14] to enable parallel execution of tiled loops. However, they do not distinguish between between intratile and intertile locality, and their experimental results are limited to small-scale multiprocessors with uniform memory access. Hence, they achieve good performance using dynamic self-scheduling with large tile sizes. In contrast, our work demonstrates that large tile sizes are not appropriate for scalable shared-memory multiprocessors with a large number of processors, and that dynamic self-scheduling performs poorly for small tiles because it does not enhance intertile locality.

Li [8] discusses a different notion of intertile reuse in an affinity tiling algorithm aimed at enhancing spatial locality for cache lines at tile boundaries. Unlike the tiling discussed in this paper, affinity tiling does not exploit temporal reuse carried by an outer loop. Reported performance improvements due to affinity tiling are limited; linear loop transformations *prior* to affinity tiling, such as loop interchange, account for most of the reported improvements. In contrast, we show that by exploiting our notion of temporal intertile reuse, we can obtain significant performance improvements.

## 6 Concluding Remarks

In this paper, we have considered scheduling of tiled loop nests to exploit wavefront parallelism on scalable shared-memory multiprocessors. We have distinguished between intratile and intertile locality to evaluate the extent to which locality is enhanced, and we have shown the impact of tile size selection on locality as well as parallelism. We have shown that dynamic self-scheduling of tiles, a common technique for exploiting wavefront parallelism, performs poorly for a large number of processors because it cannot simultaneously enhance intertile locality and provide sufficient parallelism. Failure to enhance locality is particularly detrimental to performance when a large number of processors is used because of the greater cost of accessing remote memory.

We have applied two static scheduling strategies to enhance intertile locality and overcome the shortcomings of dynamic self-scheduling. We have shown analytically that these strategies reduce the number of misses, which in turn reduces the fraction of costly misses to remote memory. Results of experiments conducted on a 30-processor Convex SPP1000 with three representative applications confirm our analysis, demonstrating that static scheduling outperforms dynamic self-scheduling by enhancing intertile locality while providing sufficient parallelism for a large number of processors. Furthermore, the performance of static block scheduling consistently improves with reductions in tile size, unlike static cyclic or dynamic scheduling. Static block scheduling improves the speedup over dynamic self-scheduling at 30 processors by factors of 1.4 for SOR, 1.8 for Jacobi, and 2.3 for LL18.

## References


[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, December 1994.

[2] G. Bell. Ultracomputers: A teraflop before its time. *Comm. of the ACM*, 35(8):26–47, August 1992.

[3] Convex Computer Corporation. *Convex Exemplar system overview*. Richardson, TX, 1994.

[4] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):814–822, October 1994.

[5] M. Heinrich et al. The Stanford FLASH multiprocessor. In *Proc. 21th Intl. Symp. on Computer Architecture*, pages 302–313, Chicago, IL., April 1994.

[6] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Comm. of the ACM*, 35(8):90–101, August 1992.

[7] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *Proc. 1993 Intl. Conf. on Parallel Processing*, pages II140–II147, August 1993.

[8] W. Li. Compiler cache optimizations for banded matrix problems. In *Proc. 1995 Intl. Conf. on Supercomputing*, pages 21–30, July 1995.

[9] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. In *Proc. 1995 Intl. Conf. on Parallel Processing*, pages II19–II28, August 1995.

[10] 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.

[11] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Computers*, 36(12):1425–1439, December 1987.

[12] Z. Vranesic et al. The NUMAchine multiprocessor. Tech. Rep. CSRI-324, Computer Systems Research Institute, University of Toronto, Canada, April 1995.

[13] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Department of Computer Science, Stanford University, 1992.

[14] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, 1991.