Improving Parallelism and Locality with Asynchronous Algorithms

Lixia Liu
Department of Computer Science
Purdue University, West Lafayette, IN 47907
liulixia@cs.purdue.edu

Zhiyuan Li
Department of Computer Science
Purdue University, West Lafayette, IN 47907
li@cs.purdue.edu

Abstract

As multicore chips become the main building blocks for high performance computers, many numerical applications face a performance impediment due to the limited hardware capacity to move data between the CPU and the off-chip memory. This is especially true for large computing problems solved by iterative algorithms because of the large data set typically used. Loop tiling, also known as loop blocking, was shown previously to be an effective way to enhance data locality, and hence to reduce the memory bandwidth pressure, for a class of iterative algorithms executed on a single processor. Unfortunately, the tiled programs suffer from reduced parallelism because only the loop iterations within a single tile can be easily parallelized. In this work, we propose to use the asynchronous model to enable effective loop tiling such that both parallelism and locality can be attained simultaneously. Asynchronous algorithms were previously proposed to reduce the communication cost and synchronization overhead between processors. Our new discovery is that carefully controlled asynchrony and loop tiling can significantly improve the performance of parallel iterative algorithms on multicore processors due to simultaneously attained data locality and loop-level parallelism. We present supporting evidence from experiments with three well-known numerical kernels.

Categories and Subject Descriptors D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.4 [**Programming Languages**]: Processors-Optimization

General Terms Algorithms, Performance

Keywords asynchronous algorithms, loop tiling, parallel numerical programs, data locality, memory performance

1. Introduction

As multicore chips become the main building blocks for high performance computers, many numerical applications face a performance impediment due to the limited hardware capacity to move data between the CPU and the off-chip memory [5][15][16]. This is especially true for large computing problems solved by iterative algorithms because of the large data set typically used.

Loop tiling, also known as loop blocking, is a program transformation technique for reducing the memory bandwidth pressure by increasing data locality. In particular, a skewed version of this technique was shown previously to be effective in enhancing the data locality for a class of iterative algorithms [3][10][11][14]. Unfortunately, the tiled programs suffer from reduced parallelism because only the loop iterations within a single tile can be easily parallelized. How to achieve parallelism and locality at the same time for iterative numerical solvers remains a challenge.

In this work, we propose to use the asynchronous model to enable effective loop tiling such that both parallelism and locality can be attained simultaneously. Asynchronous algorithms were previously proposed to reduce the communication cost and synchronization overhead between processors. Our new contribution is to show that significant performance enhancement can be achieved by combining loop tiling with carefully controlled asynchrony due to the improved data locality. We present supporting evidence from experiments with three well-known numerical kernels which were previously proven mathematically to converge under the asynchronous model.

In the rest of the paper, we first present background materials (Section 2) and the main idea (Section 3). It will be made clear that a critical issue to the success of loop tiling under the asynchronous model is to determine how often to skip the global residual tests. In Section 4, we present an adaptive scheme to make such a decision at run time. Experimental results are presented in Section 5, which is followed by a discussion of related work (Section 6). Our conclusion and remarks on future work are given in Section 7.

2. Background

In this section, we present background materials concerning the memory bandwidth issue on multicore machines, the loop tiling technique and the asynchronous model for iterative algorithms.

Multicore Architecture

Current design trend for CPUs used in both main-stream and high-performance applications is to place multiple identical computation engines (or cores) on a single CPU chip or a chip-set. On such a multicore chip or chip-set, the memory hierarchy includes multiple levels of caches. These cores connect to off-chip DRAMs through a common interface. Several CPUs can be connected to form a shared-memory multiprocessor system. Currently, the DRAMs and the CPUs are typically connected with a non-uniform memory access architecture (NUMA). As an example, Figure 1 shows the architecture of an AMD multicore-based system in which each of its four CPUs connects to a "local" DRAM. Each CPU shown in Figure 2 has four cores and a three-level cache hierarchy that includes a 64KB private L1 data cache, a

512KB private L2 cache, and a 2MB shared L3 cache (c.f. Figure 2).

When memory accesses follow regular address strides, the hardware prefetching mechanism can often accurately predict the next data to be fetched well in advance in order to hide the memory latency. Unfortunately, prefetching is effective only if memory bandwidth is sufficient to sustain the off-chip memory traffic generated by multiple cores. By now it is a well-known fact that the bandwidth available on today's multicore chips is insufficient for memory-intensive applications, including parallel programs which solve large numerical problems [5][15][16].

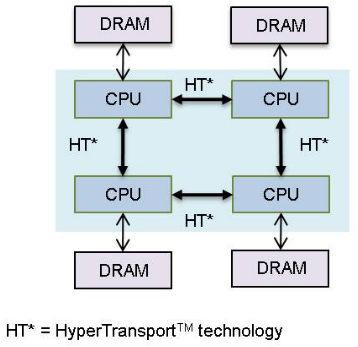


Figure 1. An example of multicore-based multiprocessors

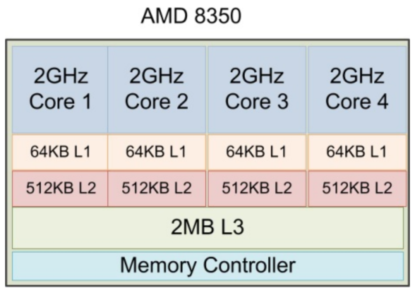


Figure 2. An example of a multicore CPU

Loop Tiling for Iterative Stencil Computation

In an iterative numerical algorithm, the same array elements are updated repetitively following a certain stencil in different time steps. A representative PDE and a code template for its iterative solver are shown in Figure 3(a-b). In this template, the step *update* computes the new value of array *a* using the previous values of some elements in array *a*. This step can be executed simultaneously on multiple processors or cores when all the dependences are satisfied.

Due to the convergence test in each time step of the iterative algorithm, a cached array element cannot be reused across different time steps unless a number of time steps are executed speculatively (i.e. before knowing whether all those steps are necessary for convergence) such that loop tiling can be performed. With such speculative execution, the maximum iteration count is partitioned into chunks such that the exit condition is tested after the execution of a chunk of *M* iterations instead of one. Each chunk of iterations is then tiled. Obviously, such a method depends on the fact that the exit condition is monotonic, i.e. if the delayed exit test fails then we know that any previous time step would have also failed the exit test.

If the semantics in the original program is strictly adhered to, the updates during the speculative execution must be buffered until a delayed convergence test shows that the speculatively executed time steps were indeed warranted, and the updates can then be committed. Furthermore, the tiles are "skewed" to satisfy all data dependences implied by the original program semantics. In the event of overshooting, a *recovery* function is invoked to roll back the execution, using the buffered values from the last checkpoint [3]. If one is further equipped with the fact that the updates are also monotonic, the recovery function can be omitted.

Problem: $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$

2D Grid $a(n,n)$

itmax: the maximum iteration count

(a) An illustrating problem

```
do t = 1,itmax
   update(a, n, f)
   ! Compute residual and convergence test
   error = residual(a, n)
   if (error .le. tol) then
      exit
   endif
end do
```

(b) The base implementation

```
do t = 1, itmax/M + 1
   ! Save the old result into buffer as checkpoint
   oldbuf(1:n, 1:n) = a(1:n, 1:n)
   ! Execute a chunk of M iterations after tiling
   update_tile(a, n, f, M)
   ! Compute residual and convergence test
   error = residual(a, n)
   if (error .le. tol) then
      call recovery(oldbuf, a, n, f)
      exit
   end if
end do
```

(c) The tiled version with speculation execution

```
do t = 1, itmax/M + 1
   ! Execute a chunk of M iterations after tiling
   update_tile(a, n, f, M)
   ! Compute residual and convergence test
   error = residual(a, n)
   if (error .le. tol) then
      exit
   end if
end do
```

(d) The tiled version without recovery

Figure 3. Tiling with speculative execution

Impact of tiling on parallelization

To this date, it remains a challenge how to effectively achieve parallelism in a tiled iterative stencil computation. This is because only the loop iterations within a single tile can be easily parallelized.

Figure 4 shows an example of tiled Jacobi with odd-even duplication, which is the best way known to tile Jacobi for data locality [3]. The 2D tile has the size of $b1 \times b2$. Prior to tiling, the step *update* can be parallelized across the entire data grid, as shown in Figure 4 (a). Each time step requires two synchronization barriers only. Unfortunately, the tiled version can be parallelized only within each tile due to data dependences between different tiles, which consist of operations on behalf of different time steps (c.f. Figure 4(b)). Hence, the tiled version suffers from small granularity of parallelism, and it increases the number of synchronization

```
!$omp parallel do
    do 20 j = 2,n-1
    do 20 i = 2,n-1
        c(i, j)=( a(i + 1, j) + a(i - 1, j) + a(i, j + 1) + a(i, j - 1) ) / 4 - f(i, j)
    20 continue
!$omp parallel do
    do 30 j = 2,n-1
    do 30 i = 2,n-1
        a(i, j) = c(i, j)
    30 continue
```

(a) subroutine update(a, n, f)

```
    do  jj = 2, n + M - 2, b1
    do  ii = 2, n + M - 2, b2
    do  t = max(1, min(jj, ii) - n + 2), min( (max(jj + b1, ii + b2) - 2), M)
        if (mod(t, 2) .eq. 1) then
!$omp parallel do
            do j = max(2, jj - t + 1), min( (jj + b1 - t), (n - 1))
            do i = max(2, ii - t + 1), min( (ii + b2 - t), (n - 1))
                c(i, j) = (a(i + 1, j) + a(i - 1, j) + a(i, j + 1) + a(i, j - 1))/4 - f(i, j)
            end do
            end do
        else
!$omp parallel do
            do j = max(2, jj – t + 1), min( (jj + b1 - t), (n - 1))
            do i = max(2, ii – t + 1), min( (ii + b2 - t), (n - 1))
                a(i, j) = (c(i + 1, j) + c(i - 1, j) + c(i, j + 1) + c(i, j - 1))/4 - f(i, j)
            end do
            end do
        end if
    end do
    end do
    end do
    if (mod(t, 2) .eq. 1) then
!$omp parallel do
        do 30 j = 2, n-1
        do 30 i = 2, n-1
            a(i, j) = c(i, j)
        30 continue
    end if
```

(b) subroutine update_tile(a, n, f, M) under the synchronous model

```
!$omp parallel
    ! Partition the grid to sub-grids based on thread ID
    partition = (n – 2) / omp_get_num_threads()
    tid = omp_get_thread_num()
    low = max(2, tid * partition)
    high = min(n-1, low + partition - 1)
    if (tid .eq. omp_get_num_threads() – 1) then
        high = n - 1
    end if
    ! Execute sub-grids asynchronously
    do  jj = low, high + M - 1, b1
    do  ii = 2, n + M - 2, b2
    do  t = max(1, min(jj – high + 1, ii - n) + 2), min( (max(jj + b1, ii + b2) - 2), M)
        if (mod(t, 2) .eq. 1) then
            do j = max(low, jj - t + 1), min( (jj + b1 - t), high)
            do i = max(2, ii - t + 1), min( (ii + b2 - t), (n - 1))
                c(i, j) = (a(i + 1, j) + a(i - 1, j) + a(i, j + 1) + a(i, j - 1))/4 - f(i, j)
            end do
            end do
        else
            do j = max(low, jj – t + 1), min( (jj + b1 - t), high)
            do i = max(2, ii – t + 1), min( (ii + b2 - t), (n - 1))
                a(i, j) = (c(i + 1, j) + c(i - 1, j) + c(i, j + 1) + c(i, j - 1))/4 - f(i, j)
            end do
            end do
        end if
    end do
    end do
    end do
    if (mod(t, 2) .eq. 1) then
!$omp do
        do 30 j = 2, n-1
        do 30 i = 2, n-1
            a(i, j) = c(i, j)
        30 continue
!$omp end do nowait
    end if
!$omp end parallel
```

(c) subroutine update_tile(a, n, f, M) underthe asynchronous model

Figure 4. Jacobi before and after tiling

barriers by a factor of the total number of tiles. Such overhead can offset the performance gain from tiling considerably, and the performance penalty increases with the number of cores.

Asynchronous Algorithms

Asynchronous algorithms have previously been proposed to reduce data communication and synchronization overhead in parallel computing [1][12][13][18][19]. The basic idea of asynchronous algorithms is to relax the data exchange requirement such that the update on each data point does not necessarily use the most up-to-date values of its neighbors. For a class of asynchronous iterative algorithms, convergence is mathematically guaranteed, although the convergence rate may potentially be slowed down due to the use of less recent updates of neighbors.

Under the asynchronous model, a processor, or a core, is allowed to start the computation of the next iteration without waiting for any other processor/core to complete the same iteration. For example, an asynchronous Jacobi method can be used to solve a system of linear equations $Ax = b$ with the solution vector x^T decomposed into q block components such that the initial vector can be written as $x_0^T = [x_0^1, \dots, x_0^q]$. Each component can be assigned to a different processor if we have q processors. At each iteration k, there may be components that are not updated. One defines the sets $J_k \subseteq \{1, 2, .., q\}$ and uses $i \in J_k$ to denote the i^{th} block component updated in the k^{th} iteration.
for $k = 1, 2, \dots$

$$x_k^i = \begin{cases} x_{k-1}^i, & i \notin J_k \\ \text{solve } A_{ii}x_k^i = b_i - \sum_{j=1, j\neq i}^{q} A_{ij}x_{r(j,k)}^i, & i \in J_k \end{cases} \quad (1)$$

The term $r(j, k)$ is used to denote the iteration number of the j^{th} block component being used in the computation of any particular block component in the k^{th} iteration. The value of $r(j, k)$ depends on the freshness of the update and how soon the update is seen by each processor. Using more up-to-date values helps accelerate the convergence. More details about asynchronous algorithms and their convergence properties can be found in [1][18][19].

In addition to the synchronization and communication benefit, the relaxed data dependences in the asynchronous model can also simplify parallel programming. Gauss-Seidel is such an example. As Figure 5 shows, in Gauss-Seidel, $a(i, j)$ (marked by the light grey circle) depends on two neighboring array elements $a(i - 1, j)$ and $a(i, j - 1)$ (dark grey circles) which are computed in the same time step. Such dependences make it difficult to directly parallelize Gauss-Seidel. To remove such dependences, one often uses a red-black update scheme which partitions the data grid into two disjoint subsets, red (circles with white dots) and black (circles without dots). These two subsets are updated in alternate turns, but the update to each set can be parallelized. The red-black Gauss-Seidel method requires more iterations to converge than the sequential Gauss-Seidel because the former uses less recent values in the computation.

Since the asynchronous model tolerates the uncertainty concerning whether the latest neighboring values are used in the update, one can directly parallelize the step *update* by partitioning the data grid into a number of sub-grids. A number of iterations (i.e. time steps) are executed over each sub-grid without synchronizing with the operations on the other sub-grids. After a number of such *inner iterations*, a barrier is set up before the step *residual* which performs a global convergence test. The asynchronous

Gauss-Seidel is superior to the red-black scheme due not only to its reduced synchronization but also to its higher likelihood to use the up-to-date neighboring elements than the latter.

3. Enabling Loop Tiling with Asynchrony

To date, the focus of asynchronous algorithm development is on the reduction of communication and synchronization overhead, but not on improving data locality. To see how we can exploit the asynchronous model to increase data locality through loop tiling, we point out three advantages offered by the asynchronous model, namely the relaxed data dependences between neighboring array elements, the exploitation of the monotonic exit condition, and furthermore, the reduced number of global residual tests. This last advantage is the result of an optimistic assumption that, by skipping some of the convergence tests, which helps reduce the communication cost, we do not risk executing too many, if any, extra iterations.

Once we accept the idea of skipping a number of global tests, we can introduce loop tiling into an asynchronous algorithm by further partitioning the sub-grid into tiles and applying inner iterations to each tile successively. With a properly chosen tile size, data once fetched to a cache of interest (e.g. the L1 cache) can be reused as often as the number of skipped global tests. For convenience, we denote the number of inner iterations applied to each tile the *chunk size*. One wants the chunk size to equal the number of time steps between two consecutive remaining global tests in order to obtain the greatest performance gain from tiling. Figure 4 (c) shows an asynchronous Jacobi algorithm with tiling.

Analyzing the Impact of the Chunk Size

As our experimental data (in Section 5) will show, the chunk size has a significant impact on the effectiveness of the tiled program under the asynchronous model. This is because a size chosen too small will reduce data reuse, but a size chosen too large will increase iteration overshooting. In order to examine the impact of the chunk size, it is useful to start by analyzing the execution time of a tiled program for a given chunk size. The execution time of the original program (T^{org}) and the *kernel* of a tiled version (T_t^{kernel}) can each be represented by a linear function in terms of k, the number of iterations:

$$T^{\text{org}} = a_o \times k + b_o \quad \text{and} \quad T_t^{\text{kernel}} = a_t \times k + b_t$$

Take Jacobi as an example, on an AMD "Barcelona" processor, we obtain $a_o = 0.23$ *and* $b_o = 0.16$. Furthermore, with a certain tile size, we obtain $a_t = 0.08$ and $b_t = 0.44$ for the tiled version. We plot the time measured for different iteration counts in Figure 6. Given the chunk size (C), the kernel of the tiled code will be executed $\left\lceil \frac{I}{C} \right\rceil$ times. The execution time of the tiled version can therefore be written as

$$T_t = \left\lceil \frac{I}{C} \right\rceil (a_t C + b_t) \quad \text{compared to} \quad T^{\text{org}} = a_o I + b_o$$

where I is the total number of iterations taken to pass the convergence test.

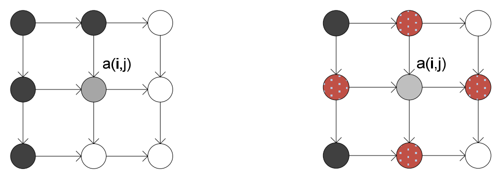
We use Δ to represent the number of extra iterations caused by overshooting in the tiled version.

$$\Delta = \left(\left\lceil \frac{I}{C} \right\rceil \times C\right) - I, \ \ 0 \le \Delta < C$$

So,

```
     do 20 j = 2,n - 1
     do 20 i = 2,n - 1
        a(i, j)=(a(i+1, j)+a(i-1, j)+a(i, j+1)+a(i, j-1))/4 - f(i, j)
     20 continue
```

(a) subroutine update(a, n, f)



(b) data flow of GS *(c) data flow of red-black GS*

Figure 5. Gauss-Seidel kernel and data flow

$$T_t = a_t(I + \Delta) + \left(\left\lceil \frac{I}{C} \right\rceil\right) b_t$$

The speedup due to tiling, $\frac{T^{\text{org}}}{T_t}$, is maximized when the chunk size $C = I$.

It is more difficult to develop a cost model for tiled code under the synchronous model due to the skewed tile shapes. Nonetheless, it is easy to see that the optimum chunk size for tiling without recovery will be the same as the one for tiling under the asynchronous model. Further, we can approximate the time of tiled version with recovery by

$$T^{\text{rec}} = \left\lfloor \frac{I}{C} \right\rfloor (a_t C + b_t) + (a_o(\mathrm{I} - \left\lfloor \frac{I}{C} \right\rfloor C) + b_o)$$

$$= a_t(I + \Delta) + \left(\left\lfloor \frac{I}{C} \right\rfloor\right) b_t + (C - \Delta) a_o + b_o$$

To minimize T^{rec} is to find $C = C^*$ which minimizes

$$\text{Cost}(C) = \Delta\, a_t + \left\lfloor \frac{I}{C} \right\rfloor b_t + (C - \Delta) a_o$$

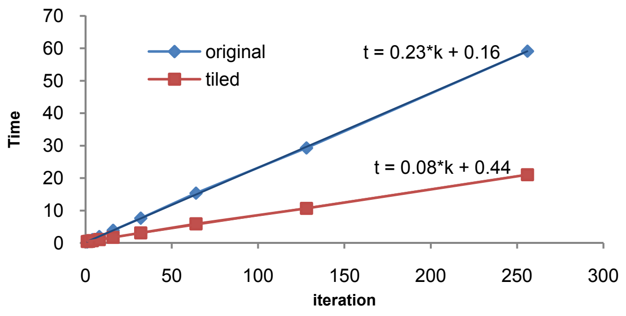


Figure 6. Fitting of the time model for the original Jacobi and the kernel of a tiled version

We determine the optimum chunk size C^* by minimizing the cost function. To make the cost function differentiable, we approximate by replacing $\left\lfloor \frac{I}{C} \right\rfloor$ with $\frac{I}{C}$, the cost function then has the differential of

$$\frac{d\,\text{Cost}(C)}{dC} \approx -\frac{I \cdot b_t}{C^2} + a_o \qquad (2)$$

which has the root

$$C^* = \sqrt{I \frac{b_t}{a_o}} \qquad (3)$$

Due to our earlier approximation of $\left\lceil \frac{I}{C} \right\rceil$ by $\frac{I}{C}$, the root derived above will be an underestimate. With fixed parameter I, a_o, b_o, a_t and b_t, we can exhaustively search for the exact C^* using our timing model. Figure 7 plots such exact optimums for different I and compare it against our approximated values. In our experiments (c.f. Section 5), we also exhaustively measure the performance for all possible chunk sizes. Our results show that the timing formulas developed above for T_t and T^{rec} are quite accurate and the execution time is highly sensitive to the choice of the chunk size.

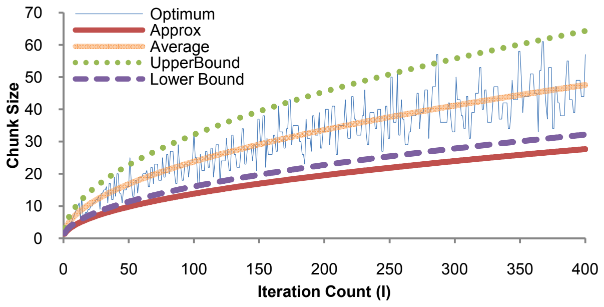


Figure 7. The optimum chunk size for tiling with recovery and its approximation

4. An Adaptive Scheme for Global Test

The discussion in the previous section makes it clear that the choice of the chunk size is highly important to the performance of loop tiling under the asynchronous model. If the exact number of iterations taken to converge can be known in advance, then one can simply create a single chunk whose size equals to that number. Unfortunately, such a number is impossible to determine in advance. Although formulas exist for estimating the convergence rate of each well-known iterative numerical algorithm, such an estimate is too imprecise for the purpose of choosing the chunk size. Let $I(\text{algorithm})$ denote the number of iterations taken for a given algorithm to converge. The following formulas are well-known for Jacobi, Gauss-Seidel, and ideal SOR, respectively:

With $p = -\log(s)$ while s represents the relative residual tolerance in the convergence test. Let $J \times J$ denote the grid size,

$$I(\text{Jacobi}) \approx \frac{\ln(s)}{\ln(\rho_{\text{Jacobi}})} \approx \frac{\ln(s)}{\ln\left(1 - \frac{\pi^2}{2J^2}\right)} \approx \frac{1}{2}pJ^2$$

$$I(\text{GS}) \approx \frac{\ln(s)}{\ln(\rho_{\text{gs}})} \approx \frac{\ln(s)}{\ln\left(1 - \frac{\pi^2}{J^2}\right)} \approx \frac{1}{4}pJ^2$$

$$I(\text{SOR}) \approx \frac{\ln(s)}{\ln(\rho_{\text{sor}})} \approx \frac{\ln(s)}{2\ln\left(\frac{\rho_{\text{Jacobi}}}{1 + (1 - \rho_{\text{Jacobi}}^2)^{1/2}}\right)} \approx \frac{1}{3}pJ$$

For example, with $s = 0.02$ and $J = 4000$, we have $I(\text{Jacobi}) = 1.36E7$, $I(\text{GS}) = 6.80E6$ and $I(\text{SOR}) = 2.27E3$. Although such formulas are useful for comparing the convergence rate among different algorithms, the actual number of iterations required for convergence when executing each algorithm can be much smaller. For example, we have measured the following numbers using the same value of s and J, $I(\text{Jacobi}) = 159$, $I(\text{GS}) = 77$, and $I(\text{SOR}) = 51$.

Without a promising method to determine the total number of iterations in advance, we have devised an adaptive scheme to determine the size of each chunk before the next global residual test. This is described as follows.

To determine the chunk size adaptively at run time, we assume that the program converges in a similar rate to the adjacent iteration. Let ρ represent convergence rate at k^{th} step with the chunk size as C, and let the residual error in the previous step and the current step be r_{k-1} and r_k respectively. We can derive ρ from the following equation.

$$\rho^C \times r_{k-1} = r_k$$

$$\therefore \log(\rho) = \frac{\log\left(\frac{r_k}{r_{k-1}}\right)}{C}$$

Thus, to pass the convergence test ($r < \text{tol}$), we need $N \geq \log\left(\frac{\text{tol}}{r_k}\right) / \log(\rho)$ more iterations to make $\rho^N \times r_k \leq \text{tol}$.

Subsequently, we select the next chunk size as the predicted minimum number of iterations required to pass the convergence test. We repeat this process for each chunk of iterations.

$$C^* = \log\left(\frac{\text{tol}}{r_k}\right) \times C \Big/ \log\left(\frac{r_k}{r_{k-1}}\right) \quad \text{then} \quad C^* \to C$$

The advantage of this adaptive scheme is that we do not need to know the number of iterations taken to converge in advance. Our experimental results (in Section 5) will show that the scheme works quite well. Figure 8 shows the code template of this adaptive scheme.

```
old_error = init_error
new_M = initial_chunk
t = 0

do while (t .le. itmax)
    t = t + new_M

    ! Execute a chunk of new_M iterations with tiling
    update_tile(a, n, f, new_M)

    ! Compute the residual and perform convergence test
    error = residual(a,n)
    if (error .le. tol) then
        exit
    else
        ! Predict the next chunk size adaptively with minimal chunk size as 'min_chunk'
        old_M = new_M
        new_M = int(log(tol / error) * old_M / log(error / old_error))
        new_M = max(min_chunk, new_M)
        old_error = error
    end if

end do
```

Figure 8. Code template of the adaptive scheme

5. Experimental Evaluation

Experimental Setup

Three different hardware platforms are used for evaluation of the effectiveness of our approach: (1) machine A, a quad-socket 2GHz quad-core AMD Opteron 8350 “Barcelona” processors; (2) machine B, a 2.4GHz quad-core Intel Q6600; and 3) machine C, a dual-socket 2.4GHz quad-core Intel Nehalem E5530. The details of the memory hierarchy and the peak memory bandwidth (*BW*) are shown in Table 1. The same table also shows the sustained memory bandwidth *SBW* obtained by measuring the bandwidth for bulk memory copy operations. We use the compiler ***ifort v9.1*** with the `-O3 -openmp` flags to compile both the baseline code (i.e. the sequential code without tiling) and the codes optimized in various ways.

We perform experiments using three numerical kernels, Jacobi, Gauss-Seidel and successive over-relaxation (SOR) to solve a Laplace equation under the following boundary conditions.

$$a(0,y) = a(1,y) = 0, 0 \leq y \leq 1$$

$$a(x,0) = \sin(\pi x), a(x,1) = \sin(\pi x)\, e^{-x}, 0 \leq x \leq 1$$

Table 1 Memory configuration of testing system

Machine	*Model*	*L1*	*L2*	*L3*	*BW (GB/s)*	*SBW (GB/s)*
A	AMD8350 4x4 cores	64KB private	512KB private	4x2MB shared	21.6	18.9
B	Q6600 1x4 cores	32KB private	2x4MB shared	N/A	8.5	4.79
C	E5530 2x4 cores	256KB private	1MB private	2x8MB shared	51	31.5

All the experimental results are measured with a grid size of 4000×4000 and the maximum iteration count of 1000.

For each test program, we want to find out answers to the following questions:

1. Does the asynchronous and tiled version (*async_tiled*) outperform the asynchronous version without tiling (*async_base*)?
2. Does the asynchronous and tiled version outperform both versions of synchronous tiled codes, one with the recovery function (*tiled*) and the other (faster) without the recovery function (*tiled_norec*)?
3. What are the impacts of the chunk size (i.e. the number of skipped global residual tests) on the data locality and the performance of the tiled programs, both synchronous and asynchronous?
4. How effective is our adaptive scheme in choosing the chunk size?

The Jacobi Program

Jacobi is a well-known numerical kernel for iterative methods. Figure 9 illustrates the performance impact of tile size on the machine A for both sequential and parallel version. On a single core, the best performance is obtained when the tile (about 7KB) fits into the L1 cache. However, since the parallelism is constrained by the tile size, the best performance is given when the tile (about 5MB) fits in the L2 cache. Figure 12 shows the impact of the tile size on all three numerical codes transformed by various tiling methods under both the synchronous and the asynchronous models.

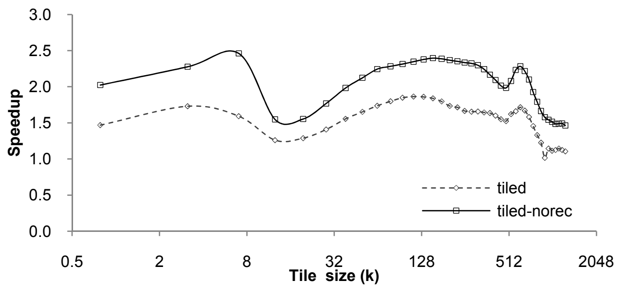
From Figure 12, we obtain the best tile size for each tiling method on each machine. Using such best tile sizes, we compare the performance between the synchronous and the asynchronous versions in Figure 10. Furthermore, we summarize the best performance for each version in Table 2. In this table (as well as Tables 3-8), the label "parallel" is for the original code parallelized under the synchronous model without tiling. For all three machines, the asynchronous tiled version *async-tiled* improves the performance significantly over the *async-base* version. This underlines the importance of data locality. On machine B and C, the performance level is similar between *async-tiled* and *tiled-norec*. However, on machine A, the asynchronous tiled version shows a clear performance advantage. The main reason is due to the limited parallelism in the synchronous versions (which is restricted to the L2 cache size). Compared to machine B and C, machine A has a relatively small L2 cache size. Asynchronous algorithm supports a higher degree of parallelism and therefore a better performance. The highest performance speedup of asynchronous version is up to 39x while those two synchronous versions can only achieve 16x and 27x respectively.

Table 2 Summary of the best performance of each version

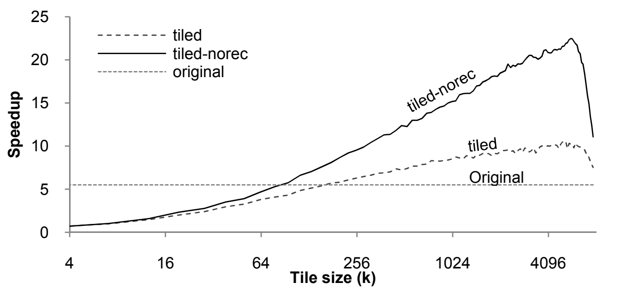
Machine	*kernel*	*parallel*	*tiled*	*tiled-norec*	*async-base*	*async-tiled*
A 16 cores	Jacobi	5.95	16.76	27.24	5.47	39.11
B 4 cores	Jacobi	1.01	2.55	3.44	1.01	3.67
C 8 cores	Jacobi	3.73	8.53	12.69	3.76	13.39

Table 3 DRAM accesses and cache misses on machine A

		DRAM	*l3 cache miss*	*l1 cache miss*	*l2 cache miss*
Jacobi	parallel	9.8E+09	6.4E+09	2.5E+08	1.6E+09
	tiled	1.0E+09	7.2E+08	9.4E+07	1.0E+09
	tiled-norec	2.9E+08	2.3E+08	7.4E+07	8.0E+08
	async-base	1.0E+10	6.4E+09	2.5E+08	1.6E+09
	async-tiled	3.0E+08	1.9E+08	1.9E+08	9.5E+07



(a) Single Core performance



(b) Performance on 16 cores

Figure 9. Impact of the tile size on the performance of Jacobi under the synchronous execution model

In Figure 10, the performance of each version fluctuates with different chunk sizes. This fluctuation is due to the overshooting as well as the recovery overhead if recovery is performed. The number of overshot iterations in those two asynchronous versions is shown in Figure 11. The fluctuation phenomenon agrees with the analytical result of our cost model and it underlines the importance of choosing a proper chunk size to the overall performance.

Using a performance monitoring tool called Pfmon [8], we also measured the count of DRAM accesses and cache misses at all levels in Table 3. The statistics show that data locality is improved under the asynchronous model, evidenced by the reduction in both the DRAM accesses and cache misses.

Gauss-Seidel (GS)

As discussed in Section 2, it is difficult to parallelize the sequential Gauss-Seidel directly. Consequently, a red-black Gauss-Seidel is commonly used. Figure 13 shows the speedup of tiled GS and tiled red-black GS over the original GS on a single core. Generally, red-black GS performs worse than the original GS. This is mainly because that the red-black GS converges slower than the original GS due to the fact that the former uses less recent values for updates.

Figure 14 compares the performance of synchronous tiled versions against two asynchronous versions on each machine. Table 4 summarizes the speedup values, with the optimum chunk size, for each method. Table 5 lists the measured DRAM accesses and cache misses. From these results, we see that the asynchronous tiled version out-performs *tile-norec* on all machines. The performance benefits on machine B and C can be attributed mainly to the fact that the asynchronous model, unlike the synchronous model, can be applied to GS directly, which avoids a slower convergence suffered by the red-black parallel GS. We note that the asynchronous tiled version does not improve performance as significantly on machine A and C as machine B, although it reduces the cache misses and DRAM accesses quite significantly, as shown in Table 5. The reason for this is that machine B has a considerably lower memory bandwidth than machine A and C, and hence the performance benefits from the improved locality more significantly.

Table 4 Summary of performance comparison

Machine	*kernel*	*parallel*	*tiled*	*tiled-norec*	*async-base*	*async-tiled*
A	GS	5.49	12.76	22.02	26.19	30.09
B	GS	0.68	5.69	9.25	4.90	14.72
C	GS	3.54	8.20	11.86	11.00	19.56

Table 5 DRAM accesses and cache misses on machine A

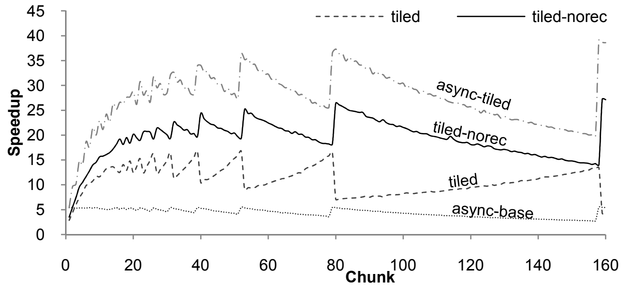
		DRAM	*l3 cache miss*	*l1 cache miss*	*l2 cache miss*
GS	parallel	8.8E+09	6.1E+09	1.2E+08	1.6E+09
	tiled	7.7E+08	5.8E+08	9.5E+07	1.0E+09
	tiled-norec	3.0E+08	2.4E+08	7.6E+07	7.6E+08
	async-base	2.0E+09	1.3E+09	2.5E+07	3.4E+08
	async-tiled	3.1E+08	2.0E+08	5.6E+07	5.9E+07

Table 6 Summary of performance comparison for SOR

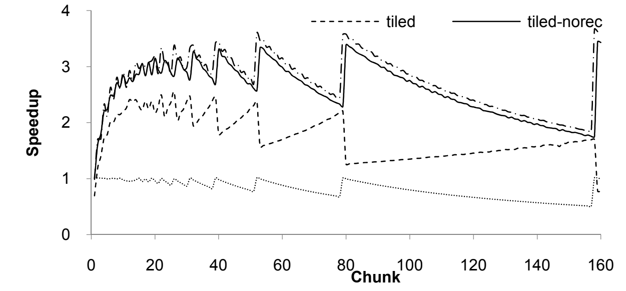
Machine	*kernel*	*parallel*	*tiled*	*tiled-norec*	*async-base*	*async-tiled*
A	SOR	4.50	11.99	21.25	29.08	31.42
B	SOR	0.65	5.24	8.54	7.34	14.87
C	SOR	3.84	7.53	11.51	11.68	19.10

Table 7 DRAM accesses and cache misses on machine A

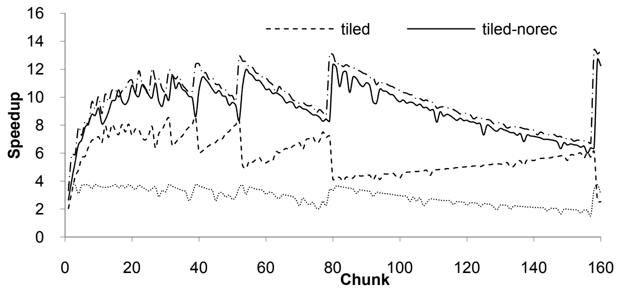
		DRAM	*l3 cache miss*	*l1 cache miss*	*l2 cache miss*
SOR	parallel	8.2E+09	5.6E+09	1.1E+08	1.4E+09
	tiled	7.0E+08	5.3E+08	1.0E+08	1.0E+09
	tiled-norec	2.6E+08	2.2E+08	7.3E+07	6.8E+08
	async-base	1.3E+09	8.8E+08	1.9E+07	2.3E+08
	async-tiled	3.3E+08	2.3E+08	4.9E+07	6.9E+07



(a) Machine A



(b) Machine B



(c) Machine C

Figure 10. Performance evaluation of multiple versions of Jacobi

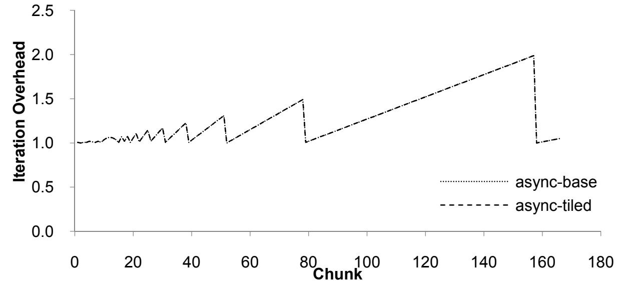
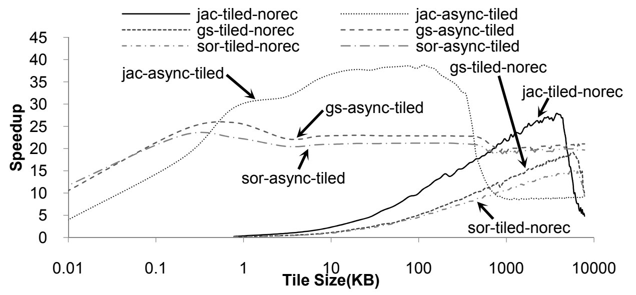


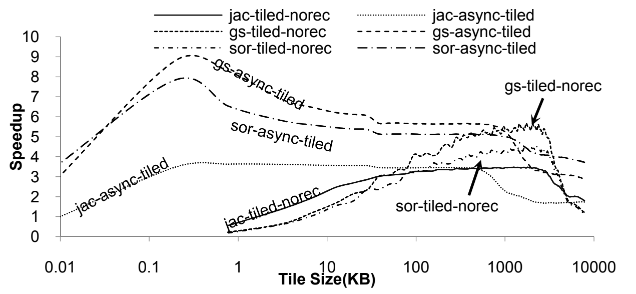
Figure 11. Overshooting overhead-more iterations (Both *async-base* and *async-tiled* are the same).

SOR

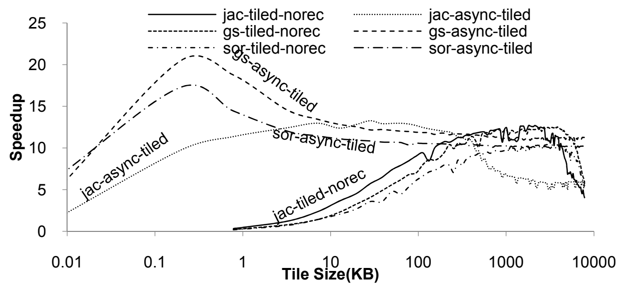
SOR has the same dependence patterns as GS. Therefore, a common way to parallelize SOR also adopts a red-black partition scheme. Figure 15 compares the performance between the red-black SOR with SOR. Figure 16 compares the performance of synchronous tiled versions against two asynchronous versions on different machines. Table 6 and Table 7 summarize the speedups and locality statistics, respectively. The performance result follows the same trend as Gauss-Seidel.



(a) Machine A



(b) Machine B



(c) Machine C

Figure 12. Best tile size on three machines

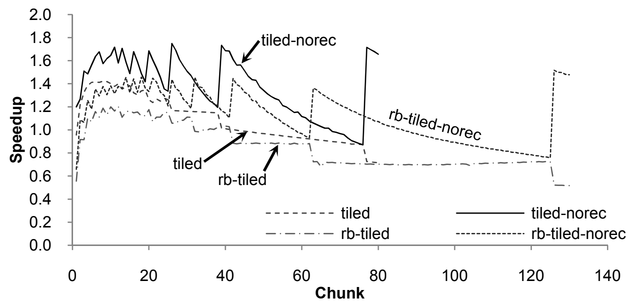
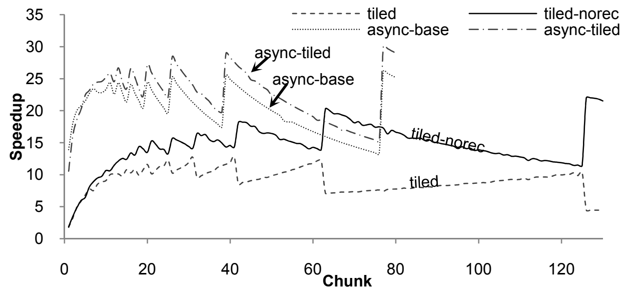
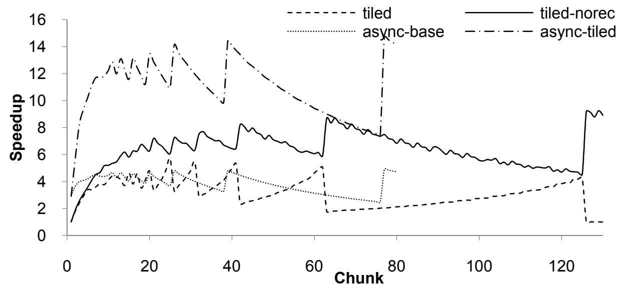


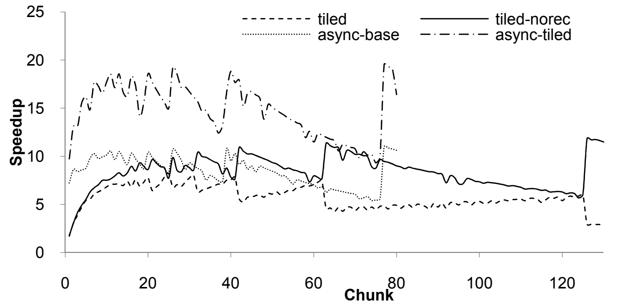
Figure 13. Evaluation of red-black Gauss-Seidel



(a) Machine A



(b) Machine B



(b) Machine C

Figure 14. Performance evaluation of Gauss-Seidel

Evaluation of Adaptive Chunk Size Selection

We applied our adaptive chunk size selection scheme to all three numerical kernels and ran experiments with two choices of the minimum chunk size, i.e. min_chunk = 1, and min_chunk = 8 (c.f. Section 4). We name these two choices adaptive-1 and adaptive-8, respectively. The result is shown in Figure 17, in which the initial chunk size is varied to see the impact. For Jacobi and SOR, the adaptive scheme achieves a performance level close to the optimal chunk size. Unlike using a fixed chunk size, the performance of adaptive selection scheme is stable and is insensitive to the initial chunk size. The only exception occurs with adaptive-1 for Gauss-Seidel, where the performance of adaptive-1 fluctuates considerably due to the difficulty of predicting the next chunk size accurately when the convergence rate starts slow down dramatically. Because a chunk size does not exploit data reuse across different iterations, we can increase the chunk size slightly to exploit data reuse such that even if we overshoot, we do not suffer much in performance. This is our rational of using adaptive-8. As Figure 17 shows, adaptive-8 exhibits good performance for all three kernels and it does not suffer instability any more. Table 8 shows the adaptive version can achieve comparable performance as the asynchronous tiled version using the optimum chunk size.

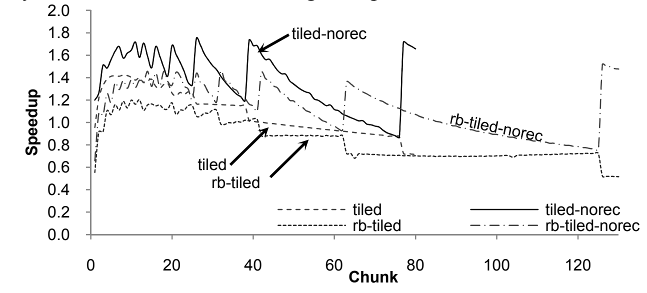
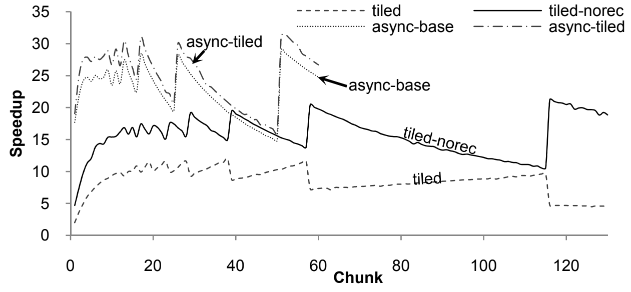
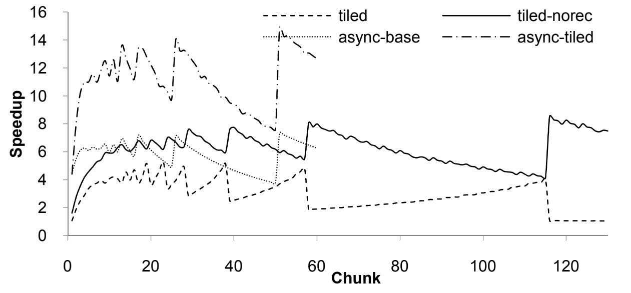


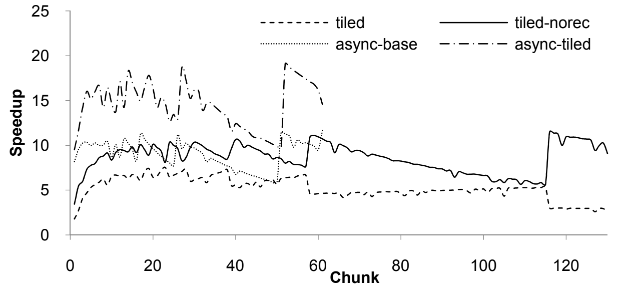
Figure 15. Evaluation of red-black SOR



(a) Machine A



(b) Machine B



(c) Machine C

Figure 16. Performance evaluation of SOR

6. Related Work

A considerable amount of prior work has been conducted on the theories of asynchronous iteration algorithms in the past of decades [12][19][20]. The evaluation of asynchronous algorithms in [1][18][21][22] shows the benefit of reducing the communication cost and synchronization overhead. It is commonly believed that multiple cores placed on the same chip or chip set will have much less communication cost and synchronization overhead when compared to processors residing on different chip sets and sockets. On the other hand, the memory bandwidth constraint on multicore is recognized as a more prominent performance bottleneck on multicore systems [5][15][16]. Our work proposes to use asynchronous model to relax dependences such that locality optimizations can be applied subsequently. To our best knowledge, this paper is the first to improve parallelism and locality simultaneously using the asynchronous model.

Compiler researchers have applied loop tiling to improve data locality for many years. Authors of [3][14] proposed skewed tiling for locality enhancement on a single processor. The loops targeted include those which are imperfectly-nested, such as the loops in Jacobi. Work in [10][11] evaluated the parallel version of such codes with tiling on multicore. Unfortunately, the tiled programs suffer from reduced parallelism because only the loop iterations within a single tile can be easily parallelized. Synchronous cache oblivious algorithms [23][24], an alternative to tiled algorithms, organize data accesses in a way to achieve locality irrespective of the cache size. Such algorithms, when successful, can naturally exploit locality at various levels of the memory hierarchy. Existing algorithms, however, have assumed the absence of a convergence test in the computation and hence are not yet suitable for the numerical problems considered in the paper. In addition, such algorithms are yet to be adapted for parallel execution.

Table 8 Summary of adaptive performance on three machines

Machine	*kernel*	*parallel*	*async-tiled*	*adaptive-1*	*adaptive-8*
A	Jacobi	5.95	39.11	29.60	29.90
	GS	5.49	28.02	13.92	26.50
	SOR	4.50	31.42	29.78	27.38
B	Jacobi	1.01	3.67	3.23	3.33
	GS	0.68	14.72	4.59	12.40
	SOR	0.65	14.87	12.13	11.81
C	Jacobi	3.73	13.39	11.69	11.86
	GS	3.54	19.56	12.15	18.12
	SOR	3.84	19.10	17.67	16.08

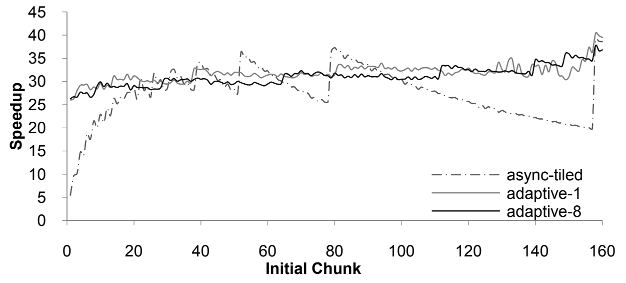
7. Conclusions and Future Work

Both parallelism and locality are important for efficient use of modern multicore CPUs. In this work, we have demonstrated the effectiveness of using the asynchronous model to relax data and control dependences and hence to enable effective loop tiling. Consequently, both parallelism and locality can be attained simultaneously. We show that how to partition the iterations into chunks has a major performance impact on tiled code and that the optimum chunk size can be derived when the total iteration count is known. On the other hand, when the total iteration count is unknown in advance, we devise an adaptive method to determine the chunk size between two consecutive global residual tests. This scheme is shown to deliver a performance level that is close to the optimum. We evaluated three well-known numerical kernels on three different systems and collected extensive statistics to examine various factors that may have an impact on the performance. The performance result shows a clear performance advantage to the approach of tiling under the asynchronous model.

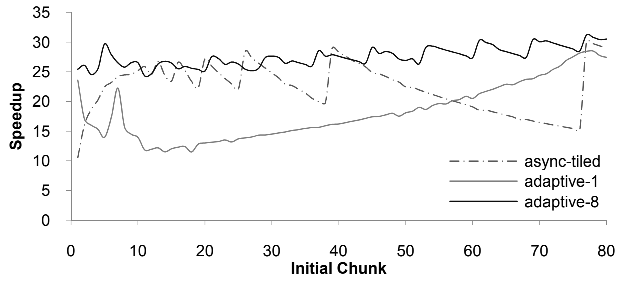
Through this work, we have addressed key issues concerning the transformation from a standard synchronous application to an asynchronous counterpart with tiling. We believe that a logical next step is to devise easy-to-use program extension or annotations based on asynchrony, such that a compiler can automatically perform the transformations described in this work. It is also important to go beyond the numerical kernels used in our experiments and investigate numerical applications which employ similar kernels.

Acknowledgment

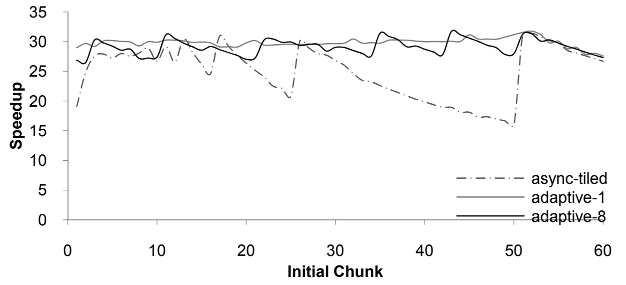
This work is sponsored in part by National Science Foundation through grants ST-HEC-0444285, CPA-0702245 and CPA-0811587, and by a Google Fellowship. The authors thank the reviewers for their careful reviews and helpful suggestions.



(a) Jacobi



(b) Gauss-Seidel



(c) SOR

Figure 17. Performance of adaptive chunk size

References

[1] Bull, J. M. Asynchronous Jacobi iterations on local memory parallel computers. M. Sc. Thesis, University of Manchester, Manchester, UK, 1991.

[2] Solving PDEs: Grid Computations, Chapter 16.

[3] Song, Y. and Li, Z. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[4] Dougals, C. C., Hasse, G., and Langer, U. A Tutorial on Elliptic PDE Solvers and Their Parallelization, SIAM.

[5] Liu, L., Li, Z., and Sameh, A. H. Analyzing memory access intensity in parallel programs on multicore. In *Proceedings of the 22nd Annual international Conference on Supercomputing*, Jun 2008.

[6] Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B. B., Garzarán, M. J., Padua, D., and von Praun, C. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 2008.

[7] SPEC benchmark. http://www.spec.org

[8] Perfmon2, the hardware-based performance monitoring interface for Linux. http://perfmon2.sourceforge.net/

[9] Jin, H., Frumkin, M., and Yan, J. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. *NAS technical report NAS-99-011*, NASA Ames Research Center.

[10] Renganarayana, L., Harthikote-Matha, M., Dewri, R., and Rajopadhye, S.. Towards Optimal Multi-level Tiling for Stencil Computations. In *Parallel and Distributed Processing Symposium,* 2007.

[11] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In *SIGPLAN Not.* 43, 6, May 2008.

[12] Frommer, A. and Szyld, D. B. Asynchronous two-stage iterative methods. In *Numer. Math.* 69, 2, Dec 1994.

[13] Meyers, R. and Li, Z. ASYNC Loop Constructs for Relaxed Synchronization. In *Languages and Compilers For Parallel Computing: 21th international Workshop,* Aug *2008.*

[14] Huang, Q., Xue, J., and Vera, X. Code tiling for improving the cache performance of PDE solvers. In *Proceedings of International Conference on Parallel Processing*, Oct 2003.

[15] Alam S. R., Barrett, B. F., Kuehn J. A., Roth P. C., and Vetter J. S. Characterization of Scientific Workloads on Systems with Multi-Core Processors. *In International Symposium on Workload Characterization*, 2006.

[16] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[17] Pugh, W., Rosser, E., and Shpeisman, T. Exploiting Monotone Convergence Functions in Parallel Programs. *Technical Report. UMI Order Number: CS-TR-3636.*, University of Maryland at College Park.

[18] Blathras, K., Szyld, D. B., and Shi, Y. Timing models and local stopping criteria for asynchronous iterative algorithms. In *Journal of Parallel and Distributed Computing*, 1999.

[19] Bertsekas, D. P. and Tsitsiklis, J. N. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings of the 3rd international Conference on Supercomputing*, 1989.

[20] Baudet, G. M. Asynchronous Iterative Methods for Multiprocessors. *J. ACM* 25, 226-244, Apr 1978.

[21] Blathras, K., Szyld, D. B., and Shi, Y. Parallel processing of linear systems using asynchronous. Preprint, Temple University, Philadelphia, PA, April 1997.

[22] Venkatasubramanian, S. and Vuduc, R. W. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international Conference on Supercomputing*, June 2009.

[23] Prokop, H. Cache-oblivious algorithms. Master's thesis, MIT, June 1999.

[24] Frigo, M. and Strumpen*,* V. The memory behavior of cache oblivious stencil computations. *J. Supercomput.* 39, 2, 93-112. Feb 2007.