

# Using Kernel Coupling to Improve the Performance of Multithreaded Applications

Jonathan Geisler  
Computing and System Sciences  
Taylor University, Upland IN 46989  
Email: jgeisler@css.tayloru.edu

Valerie Taylor, Xingfu Wu  
Department of Electrical and Computer Engineering  
Northwestern University, Evanston IL 60208  
Email: {geisler, taylor, wuxf}@ece.northwestern.edu

Rick Stevens  
Mathematics and Computer Science Division  
Argonne National Laboratory, Argonne IL 60439  
Email: stevens@mcs.anl.gov

## Abstract

Kernel coupling refers to the effect that kernel  $i$  has on kernel  $j$  in relation to running each kernel in isolation. The two kernels can correspond to adjacent kernels or a chain of three or more kernels in the control flow of an application. In previous work, we used kernel coupling to provide insights on where further algorithm and code implementation work was needed to improve performance, in particular the reuse of data between kernels. Further, coupling was used to develop analytical models of applications as a composition of the models of the kernels that make-up the application. In contrast, this paper extends the coupling concept to provide scaling in addition to coupling information about multithreaded application. We illustrate the use of the extended coupling parameter with two case studies, focused on the LU and Radix Sort of Splash-2 benchmarks executed on the Cray MTA. The results indicate up to 55% decrease in phantoms (or NOPs) and 14% decrease in execution time.

## 1 Introduction

Kernel coupling refers to the effect that kernel  $i$  has on kernel  $j$  in relation to running each kernel in isolation. The two kernels can correspond to adjacent kernels or a chain of three or more kernels in the control flow of the application. In previous work, we used kernel coupling to identify parts of the application that required performance improvements [2]. In particular, the coupling value provided insight into where further algorithm and code implementation work was needed to improve the reuse of data between kernels. Further, in [3, 8], coupling was used to develop analytical models of applications as a composition of the models of the kernels that make-up the application. In con-

trast, this paper extends the coupling concept to provide scaling in addition to coupling information about multithreaded applications.

Multithreaded applications have the interesting attribute of shared data; hence memory is the shared resource for which the coupling parameter can indicate if the sharing between kernels is good, or constructive, or if the sharing is poor, or destructive. In this paper, we focus on using the coupling parameter to provide insights into program revisions that can result in better sharing among kernels as a function of the number of threads as well as indicate when the number of threads affects the sharing.

In this paper, we illustrate the use of the coupling parameter to improve the performance of two of the Splash-2 benchmarks: Radix Sort and LU. For the case of the Radix Sort benchmark, the coupling parameter identified the communication kernel as a bottleneck, for which modifications to the synchronization scheme resulted in a reduction in phantoms (or NOPs) by 27% and a reduction in execution time by 9%. For the case of the LU benchmark, the coupling parameter indicated problems with the barrier scheme, for which modifications resulted in a 55% reduction in phantoms and a 14% reduction in execution time.

The remainder of this paper is organized as follows. In Section 2, we extend our previous work to multithreaded applications. We illustrate the use of the coupling parameter with two application in Section 3 and related work is discussed in Section 4. We summarize the paper in Section 5.

## 2 Extending the Coupling Parameter for Parallelism

Due to space concerns, we refer the reader to our previous work [8] for a description of the generalized coupling parameter.

The generalized coupling parameter does not take into account parallelization, so it must be further refined to reflect environments that can support parallel processing. There are two straightforward ways to do this:

1. Treat the number of threads as an independent entity from coupling. In this case, the numerator and denominator consider the performance of kernels using the same number of threads.

$$c_S(T) = \frac{p_S(T)}{\sum_{k \in S} p_k(T)} \quad (1)$$

$c_S(T)$  is the coupling parameter for the set of  $S$  kernels with  $T$  threads,  $p_S(T)$  is the performance of the  $S$  kernels run together with  $T$  threads,  $\mathcal{F}$  is the combining function that assumes no interaction between the kernels in  $S$ , and  $p_k(T)$  is the performance of kernel  $k$  in isolation with  $T$  threads.

2. Treat the number of threads as directly affecting coupling. In this case, the numerator must assume that the number of threads affect the overall performance of the application. We now add a scaling function  $\mathcal{E}$  to the equation so the numerator can estimate what the performance of kernel  $k$  should be with  $T$  threads in relation to 1 thread, hence  $\mathcal{E}(p_k(1), T)$  will take as input the performance of kernel  $k$  using 1 thread ( $p_k(1)$ ) and the number of threads ( $T$ ) to scale. We still require the combining function  $\mathcal{F}$  to combine the scaled performances, so the generalized equation looks like

$$c_S(T) = \frac{p_S(T)}{\sum_{k \in S} \mathcal{E}(p_k(1), T)} \quad (2)$$

where  $\mathcal{E}(x, y)$  is the model that estimates the performance of kernel  $k$  with  $y$  threads using the the serial performance ( $x$ ). Typically,  $\mathcal{E}$  will be defined as  $\mathcal{E}(x, y) = \frac{x}{y}$  since, ideally, we expect  $y$  threads to finish a task  $y$  times more quickly than 1 thread, however,  $\mathcal{E}$  may defined differently, if needed (e.g., if the application contains some kernels that must be executed sequentially, then we would define  $\mathcal{E}(x, y) = x$  for those kernels, since adding new threads is not expected to help). More

realistic  $\mathcal{E}$  models can be used if the application developer knows that the application has a particular pattern of scaling. Note that  $\mathcal{E}$  can be considered a measure of the efficiency of the  $S$  kernels.

There are four possible scenarios with respect to scaling that can occur:

1. One or more of the kernels do not scale in isolation, but when combined with the other kernels, the kernels scale well due to good coupling. Equation 1 will show good coupling with these kernels as the number of threads increase because some  $p_k$  values will overestimate the execution time. Equation 2 will show no coupling, however, because  $\mathcal{E}$  expects the kernels to scale in isolation and they aren't, so the various  $p_k$  values will correctly estimate the execution times.
2. All the kernels scale well in isolation, but due to poor coupling, scale poorly when combined. Equation 1 will show poor coupling with these kernels as the number of threads increase because some  $p_k$  values will underestimate the execution time. Equation 2, likewise, will show poor scaling since it assumes that the kernels will scale well and so the denominator will be much smaller than the numerator.
3. All the kernels scale well in isolation and also scale well when combined. Equation 1 will show no coupling for any number of threads since the kernels perform similar in isolation and combined. Equation 2, likewise, will show no coupling since the kernels perform as assumed.
4. One or more of the kernels do not scale in isolation, and when combined with other kernels, they cause them all to scale poorly. Equation 1 will show no coupling for any number of threads since the kernels perform similar in isolation and combined. Equation 2 will show poor coupling as the number of threads increase, however, since it assumes the kernels will scale when they do not.

Neither equation can capture these variations by itself. If an application scales poorly, it may be because a kernel scales poorly (Scenario 4), but it may be because two kernels do not couple well together (Scenario 2). Conversely, an application may scale well, but that may be because it uses good algorithms that scale (Scenario 3); however, it also may be because kernels coupling well in the presence of additional threads (Scenario 1). Thus, it is necessary to look at Equation 1 in conjunction with Equation 2 to determine why an application is scaling the way it is. Table 1 shows how

| Isolated | Combined   |  |
|----------|--|--|
|          | Poor   | Well   |
| Poor     | Equation 1 shows no coupling.<br>Equation 2 shows destructive coupling.          | Equation 1 shows constructive coupling.<br>Equation 2 shows no coupling. |
| Well     | Equation 1 shows destructive coupling.<br>Equation 2 shows destructive coupling. | Equation 1 shows no coupling.<br>Equation 2 shows no coupling.           |

Table 1: Scaling of kernels in isolation vs. combined with other kernels

the two equations perform for all four scenarios in a compact manner.

Using both equations allows us to look at the scaling of an application from the inside outward. Since coupling considers how various kernels within an application affect each other and the two coupling equations allow us to determine whether the scaling difficulties are due to coupling, we can see within an application to better understand why an application scales the way it does.

### 3 Experimental Results

In this section we present some experimental results of using the coupling parameter to improve the performance of two SPLASH benchmarks, Radix Sort and LU, executed on the Cray MTA architecture.

#### 3.1 Cray MTA

The Cray MTA architecture [1] is unique in that it explicitly supports threads in the hardware. The architecture was designed to tolerate the extreme memory latencies that exist in modern architectures. To improve scalability and simplify the design, the Cray MTA architecture does not contain data caches. Its 21-stage pipeline must be filled by different threads to achieve high performance execution.

The role of creating and scheduling threads on the MTA lies mostly in the compiler’s domain. It must estimate the efficiency and utilization of the processor by the program being compiled and produce machine code that will maximize these two parameters. Loops are most easily parallelized and the user can provide compiler hints to help the compiler do the best possible job. In addition to compiler hints, MTA has created ways for the program writer to explicitly create threads for coarse grain parallelism.

At each processor cycle, the MTA hardware will run any thread provided:

- The thread does not have an instruction already in the 21 cycle pipeline. This means that a serial program can only use 1/21 of the processor power waiting for each instruction to complete the pipeline before the next starts.

- All the data the instruction accesses has been loaded into a register. If the instruction is waiting on a load, it will not be available until the memory instruction completes. A typical memory instruction takes 150 cycles to retrieve, so if an entire application consisted of one load for every arithmetic operation, it would need to consist of 150 threads to efficiently use the resources, but the hardware only supports a maximum of 128 threads. This is not too strenuous of a constraint since most applications perform much more arithmetic than memory accesses.
- If the above are satisfied, the processor will try to schedule the remaining threads in some sort of fair order to prevent one thread from using all the processor resources at another’s expense.

When the processor cannot find a thread that satisfies all of these criteria, it does not perform any operation and terms this a “phantom” cycle. One of the keys to efficient execution of the MTA is minimizing the number of phantom cycles by providing enough threads so that the processor always has one ready to execute. Thus, the performance measurements will focus on this metric.

Since the processor only has one set of functional units, a serial application that fully utilizes the processor can only see a maximum improvement of 21 times. This leads us to define the efficiency component of the coupling parameter to be  $\mathcal{E}(x, y) = \frac{x}{\min(y, 21)}$ . There are many kernels that will see improvements larger than 21 because they don’t use the processor efficiently with only 1 thread, and the coupling parameter will take this into account.

#### 3.2 Splash-2 Case Studies

The Splash-2 benchmarks [9] were designed to measure the performance of shared memory processors. The MTA is a shared memory machine since every thread accesses the same memory no matter which processor it is running on. However, the codes were written to minimize data movement between processors that didn’t share caches. We can use this benchmark as a good starting point for performance measurement and using coupling to indicate which portions of the code

need to be updated for the different architecture. The MTA marketing materials suggest that this is a recommended method for porting code to this architecture [1]: start with the serial version and rely on the good parallelizing compiler to take care of the rote work so you can concentrate on the bottlenecks.

### 3.2.1 Case Study 1: Radix Sort

The initial code performs a standard radix sort by partitioning the array to be sorted. Each thread counts the keys in its partition in  $O(k/t)$  steps and then communicates its data with the other threads using a tree-based scheme. Each level of the tree is traversed synchronously so that it takes  $O(\log t \times r)$  steps. This is not optimal in an algorithmic sense, since the communication could take place in  $O(\log t + r)$  steps using fine-grain synchronization. Most shared memory machines, however, would suffer serious performance degradation trying to use such synchronization, so the code uses the coarser-grain size of the tree levels. Finally, each thread places the keys it owns into the correct location in the new array in  $O(k/t)$  steps.

We split the application into four kernels:

1. *Local Histogram* computes a histogram of the values stored by each thread. This is completely parallel with no dependencies and should already be as efficient as possible.
2. *Gather* computes a global histogram of the values stored by all the threads communicating the data up a tree-like structure.
3. *Scatter* returns the global histogram to all the threads in the reverse order it was gathered in the previous step.
4. *Arrange* moves the items each thread owns to its correct position in the newly sorted array. If the *gather* and *scatter* operations are done intelligently, there are no dependencies between threads in this kernel as each thread knows where to send the data without consulting other threads. This also means that *arrange* relies on *gather* and *scatter* returning “correct” data and cannot be run separately.

For Table 2, the first column contains the kernels being run for the measurements in the following columns. The rest of the columns list the number of phantoms that occur for the corresponding number of threads and set of kernels. For example, there are 8,431,699 phantoms when we measure the scatter kernel alone with two threads and 230,834,134 phantoms when we measure the histogram, gather, and scatter kernels together with sixteen threads.

Tables 3 and 4 show the coupling values computed from Table 2 using Equations 1 and 2 respectively. The first column lists the kernel couplings being measured, and the rest of the columns list the coupling for the number of threads and the kernels. For example, the coupling for the gather and scatter kernels with four threads is 0.983 when not considering  $\mathcal{E}$  (Equation 1) and is 8.995 when considering  $\mathcal{E}$  (Equation 2).

As we can see from Tables 3 and 4,  $c_S$  is very large when  $S = \{\textit{gather}, \textit{scatter}\}$  and when we use Equation 2 (with the scaling factor  $\mathcal{E}$ ), but does quite well when using Equation 1. This indicates that these two kernels do not scale well individually, but still manage to help each other when run together. If we can find a version of *gather* and *scatter* that scale better individually and maintain good coupling, we hope to see improvement in the overall execution. These are the two kernels that are not trivially parallel, so they can be changed to fit the architecture better.

Since the coupling values for the entire application using Equation 2 are slightly worse than the coupling values using Equation 1, we see that the scaling of the entire application could improve slightly, but it isn’t too bad considering  $c_S$  with  $\mathcal{E}$  starts at 2.785 and decreases to 1.177. This is because the kernels couple well together as more threads are added since  $c_S$  without  $\mathcal{E}$  starts at 2.785 and drops to 1.094. We can anticipate that much of the scaling decrease can be attributed to the poor scaling in *gather* and *scatter*.

Table 5 gives the analogous information to Table 2 with the improvements made to the radix code that improve the gather and scatter kernels. The coupling parameter immediately identified the communication kernels as having the worst coupling. Since the MTA supports very fine-grain synchronization using a full/empty bit for every machine word, the code was changed to the more efficient solution and resulted in 15-27% less phantoms for 32 and 64 threads. There was a significant performance penalty for 1 and 2 threads (15% more phantoms) that equalized by 8 threads (2% less phantoms). This penalty occurs from the extra overhead of the fine-grained locks that do not help with small numbers of threads.

Tables 6 and 7 are the computed coupling parameters from Table 5 using Equations 1 and 2 respectively. We can see that the scaling of radix, while still not very good, improved greatly from the original version. This improvement lead to the decrease in phantoms for large numbers of threads. This decrease in phantoms lead to an improvement in execution time of up to 9% over the original. The coupling for the untouched kernels remains good with  $\mathcal{E}$ , while the *gather* and *scatter* kernels improved.

| $S$                                   | number of threads |         |         |         |       |       |       |
|---------------------------------------|-------------------|---------|---------|---------|-------|-------|-------|
|                                       | 1                 | 2       | 4       | 8       | 16    | 32    | 64    |
| {histogram}                           | 3,062.5           | 1,406.6 | 766.5   | 382.3   | 157.6 | 105.4 | 58.0  |
| {gather}                              | 6.8               | 8.6     | 20.5    | 16.2    | 23.0  | 32.2  | 48.7  |
| {scatter}                             | 6.6               | 8.4     | 10.0    | 16.2    | 24.7  | 43.8  | 50.9  |
| {histogram, gather}                   | 3,258.2           | 1,579.6 | 766.1   | 330.6   | 195.7 | 121.0 | 69.9  |
| {gather, scatter}                     | 9.3               | 12.6    | 30.0    | 35.5    | 42.1  | 51.9  | 72.2  |
| {histogram, gather, scatter}          | 3,270.5           | 1,608.8 | 698.3   | 366.3   | 230.8 | 129.1 | 125.1 |
| {histogram, gather, scatter, arrange} | 8,565.0           | 4,527.3 | 1,953.5 | 1,123.8 | 567.5 | 286.2 | 172.4 |

Table 2: Original radix phantoms (in millions)

| $S$                                   | number of threads |       |       |       |       |       |       |
|---------------------------------------|-------------------|-------|-------|-------|-------|-------|-------|
|                                       | 1                 | 2     | 4     | 8     | 16    | 32    | 64    |
| {histogram, gather}                   | 1.062             | 1.116 | 0.973 | 0.830 | 1.084 | 0.879 | 0.655 |
| {gather, scatter}                     | 0.700             | 0.738 | 0.983 | 1.095 | 0.884 | 0.682 | 0.725 |
| {histogram, gather, scatter}          | 1.063             | 1.130 | 0.876 | 0.883 | 1.125 | 0.712 | 0.794 |
| {histogram, gather, scatter, arrange} | 2.785             | 3.180 | 2.451 | 2.710 | 2.765 | 1.578 | 1.094 |

Table 3: Original radix coupling using the Generalized Equation (Equation 1)

| $S$                                   | number of threads |       |       |        |        |        |         |
|---------------------------------------|-------------------|-------|-------|--------|--------|--------|---------|
|                                       | 1                 | 2     | 4     | 8      | 16     | 32     | 64      |
| {histogram, gather}                   | 1.062             | 1.029 | 0.998 | 0.862  | 1.020  | 0.828  | 0.478   |
| {gather, scatter}                     | 0.699             | 1.883 | 8.995 | 21.261 | 50.553 | 81.639 | 113.702 |
| {histogram, gather, scatter}          | 1.063             | 1.046 | 0.908 | 0.953  | 1.201  | 0.881  | 0.854   |
| {histogram, gather, scatter, arrange} | 2.785             | 2.944 | 2.540 | 2.923  | 2.952  | 1.954  | 1.177   |

Table 4: Original radix coupling using the Scaling Equation (Equation 2)

| $S$                                   | number of threads |         |         |         |       |       |       |
|---------------------------------------|-------------------|---------|---------|---------|-------|-------|-------|
|                                       | 1                 | 2       | 4       | 8       | 16    | 32    | 64    |
| {histogram}                           | 2,892.6           | 1,437.1 | 795.0   | 394.6   | 204.1 | 85.3  | 48.5  |
| {gather}                              | 3.6               | 7.7     | 6.5     | 9.3     | 9.1   | 11.1  | 10.7  |
| {scatter}                             | 3.6               | 7.7     | 6.5     | 9.3     | 9.1   | 11.1  | 10.7  |
| {histogram, gather}                   | 3,253.8           | 1,584.1 | 824.3   | 379.5   | 177.2 | 102.8 | 38.6  |
| {gather, scatter}                     | 5.1               | 12.1    | 12.3    | 21.3    | 19.7  | 28.1  | 24.4  |
| {histogram, gather, scatter}          | 3,217.8           | 1,549.0 | 746.5   | 374.5   | 224.4 | 111.8 | 50.2  |
| {histogram, gather, scatter, arrange} | 9,904.5           | 5,252.9 | 2,592.6 | 1,105.1 | 614.7 | 245.2 | 126.1 |

Table 5: Improved radix phantoms

| $S$                                   | number of threads |       |       |       |       |       |       |
|---------------------------------------|-------------------|-------|-------|-------|-------|-------|-------|
|                                       | 1                 | 2     | 4     | 8     | 16    | 32    | 64    |
| {histogram, gather}                   | 1.134             | 1.096 | 1.028 | 0.940 | 0.831 | 1.065 | 0.653 |
| {gather, scatter}                     | 0.560             | 0.785 | 0.800 | 1.055 | 0.921 | 0.822 | 0.670 |
| {histogram, gather, scatter}          | 1.109             | 1.066 | 0.921 | 0.903 | 0.996 | 0.935 | 0.592 |
| {histogram, gather, scatter, arrange} | 3.413             | 3.616 | 3.199 | 2.664 | 2.727 | 2.051 | 1.486 |

Table 6: Improved radix coupling using the General Equation (Equation 1)

| $S$                                   | number of threads |       |       |        |        |        |        |
|---------------------------------------|-------------------|-------|-------|--------|--------|--------|--------|
|                                       | 1                 | 2     | 4     | 8      | 16     | 32     | 64     |
| {histogram, gather}                   | 1.134             | 1.094 | 1.138 | 1.048  | 0.979  | 0.746  | 0.280  |
| {gather, scatter}                     | 0.560             | 2.634 | 5.340 | 18.518 | 34.224 | 64.275 | 55.700 |
| {histogram, gather, scatter}          | 1.109             | 1.068 | 1.029 | 1.032  | 1.238  | 0.809  | 0.364  |
| {histogram, gather, scatter, arrange} | 3.413             | 3.620 | 3.574 | 3.047  | 3.389  | 1.775  | 0.913  |

Table 7: Improved radix coupling using Scaling Equation (Equation 2)

### 3.2.2 Case Study 2: LU

The LU kernel performs LU factorization on a dense matrix. This factorization involves transforming the matrix  $A$  into two matrices: a lower diagonal matrix  $L$  and an upper diagonal matrix  $U$  such that  $A = LU$ . Each row depends on all the previous rows in the matrix to be computed before it can be. The original code partitions the matrix into blocks with each thread responsible for some portion of the matrix. Synchronization between rows is done through barriers; we have identified `lu0()`, `bdiv()`, `bmod()`, and `bmodd()` as the four primary kernels in the inner loop of the factorization.

This means that initially, only one thread is active and the others are waiting for its results. In the next step, one row of blocks and one row of columns are active and the rest are waiting for their results. The last step of the inner loop can be done in parallel with the first step of the next iteration of the loop, but the diagonal block must have two operations performed on it, so there is a large load imbalance toward the thread processing the diagonal. Finally, once a thread has performed an operation on its data as the main diagonal, it no longer participates in the computation, resulting in lessening efficiencies as the computation progresses.

Tables 9 and 10 contain the coupling parameters from the measurements in Table 8 using Equations 1 and 2 respectively. The coupling parameter shows poor scaling (especially with 64 threads) in all combinations of the kernels, but those with both `bdiv` and `bmod` did the worst. Without the  $\mathcal{E}$  scaling function, the kernels couple constructively most of the time. Using Table 1 as our guide, we can see that our code doesn't fit nicely into any of the four categories. Equation 1 shows constructive coupling, indicating the kernels do not scale as well in isolation as they do together. Equation 2 indicates that even though the kernels couple well together, they still do not scale well. If we can remove the problem with scaling, we can hope to see better runtime since the kernels benefit well from each other.

We begin by looking at the most probable cause for poor scaling. By looking at the algorithm, we see that the barriers are the most obvious causes for potential inefficiency. If we can come up with an algorithm that allows all threads to execute without waiting for other threads to complete, we have a much better chance of keeping the processor pipeline full. The original algorithm kept each data item tightly associated with a specific processor to best utilize the cache coherent architecture it was developed on; however, the MTA may move data freely between processors because it does not contain any local data caches: any thread on

any processor can process a computable element.

With this in mind, we dynamically assigned available threads the next row in the matrix to be processed. Since the size of the matrix grows faster than the number of threads, there is very little chance for a thread to be stalled waiting for another thread to finish. This can only happen if a thread is assigned row  $X$  and another thread is assigned row  $X$  in the next iteration of the elimination, and that will most likely happen when the number of rows to compute is less than the number of available threads near the end of the computation. This inefficiency is assumed to be a negligible portion of the runtime and ignored in this context.

Since the algorithm is completely different, we do not include the kernels from the previous algorithm for coupling comparison, but rather include the total phantom (Table 11) and runtime (Table 12) information.

The data shows that we have improved the scaling of LU in the improved code. The number of phantoms are reduced by 55% for 64 threads with an overall improvement in execution by 14%. There is a larger amount of overhead associated with the new code since there are more locks and synchronization points that are accessed that do not benefit the application with only a few threads executing. The single thread case gives a 51% degradation in phantoms and execution time. This is nearly equalized by 16 threads when the old code has 2% more phantoms and 11% more execution time. For 32 and 64 threads, we see the improvement.

We have used performance measurement tools for the MTA to verify that the processor pipeline is more consistently full using the new algorithm. The original algorithm shows a cyclical pattern depending on which phase of the loop is being executed while the new algorithm shows a more constant (and efficient) usage of the pipeline. The removal of the BARRIERS greatly improved the efficiency and allowed the computation to be finished more quickly with sufficient number of threads.

## 4 Related Work

Allan Snively defined a concept similar to coupling called symbiosis [7]. His work studied the interaction of two separate programs on a multithreaded machine; in contrast our work focuses on the interactions within a single application. The equations used to quantify interaction uses a similar ratio of isolated and simultaneous execution times. His work in [6] indicated that dynamic scheduling of multiple highly tuned applications still produces better efficiency on the Cray MTA.

| $S$                      | number of threads |          |          |          |          |         |          |
|--------------------------|-------------------|----------|----------|----------|----------|---------|----------|
|                          | 1                 | 2        | 4        | 8        | 16       | 32      | 64       |
| {lu0}                    | 375.7             | 404.6    | 885.3    | 587.4    | 374.2    | 361.2   | 4,445.6  |
| {bdiv}                   | 641.2             | 551.4    | 810.9    | 818.4    | 400.8    | 367.5   | 2,238.2  |
| {bmodd}                  | 118,239.2         | 71,814.9 | 44,194.8 | 28,863.7 | 14,169.3 | 6,445.7 | 17,420.8 |
| {lu0, bdiv}              | 652.7             | 535.7    | 953.6    | 710.9    | 422.3    | 371.1   | 1,671.6  |
| {bdiv, bmod}             | 894.8             | 859.0    | 380.7    | 950.9    | 518.2    | 384.7   | 1,634.5  |
| {lu0, bdiv, bmod}        | 919.5             | 811.1    | 1,134.9  | 901.4    | 529.7    | 423.3   | 1,987.6  |
| {bdiv, bmod, bmodd}      | 112,527.8         | 71,584.5 | 46,123.9 | 29,840.8 | 15,363.9 | 5,972.9 | 40,347.2 |
| {lu0, bdiv, bmod, bmodd} | 113,962.0         | 69,135.9 | 46,548.0 | 29,958.7 | 15,697.0 | 6,138.3 | 5,442.1  |

Table 8: Original LU phantoms (in millions)

| $S$                      | number of threads |       |       |       |       |       |       |
|--------------------------|-------------------|-------|-------|-------|-------|-------|-------|
|                          | 1                 | 2     | 4     | 8     | 16    | 32    | 64    |
| {lu0, bdiv}              | 0.642             | 0.560 | 0.562 | 0.506 | 0.545 | 0.509 | 0.250 |
| {bdiv, bmod}             | 1.395             | 1.558 | 0.469 | 1.162 | 1.293 | 1.047 | 0.730 |
| {lu0, bdiv, bmod}        | 0.904             | 0.848 | 0.669 | 0.641 | 0.683 | 0.581 | 0.297 |
| {bdiv, bmod, bmodd}      | 0.947             | 0.989 | 1.025 | 1.005 | 1.054 | 0.877 | 2.052 |
| {lu0, bdiv, bmod, bmodd} | 0.956             | 0.950 | 1.014 | 0.990 | 1.050 | 0.856 | 0.226 |

Table 9: Original LU coupling using the Generalized Equation (Equation 1)

| $S$                      | number of threads |       |       |        |        |        |        |
|--------------------------|-------------------|-------|-------|--------|--------|--------|--------|
|                          | 1                 | 2     | 4     | 8      | 16     | 32     | 64     |
| {lu0, bdiv}              | 0.956             | 1.159 | 1.561 | 2.010  | 2.106  | 1.081  | 0.958  |
| {bdiv, bmod}             | 1.395             | 2.679 | 2.375 | 11.864 | 12.930 | 12.599 | 53.484 |
| {lu0, bdiv, bmod}        | 0.904             | 1.595 | 4.464 | 7.091  | 8.334  | 8.741  | 41.045 |
| {bdiv, bmod, bmodd}      | 0.947             | 1.204 | 1.552 | 2.008  | 2.068  | 7.664  | 34.519 |
| {lu0, bdiv, bmod, bmodd} | 0.956             | 1.159 | 1.561 | 2.010  | 2.106  | 1.081  | 0.958  |

Table 10: Original LU coupling using the Scaling Equation (Equation 2)

| number of threads | original phantoms | improved phantoms |
|-------------------|-------------------|-------------------|
| 1                 | 15,741.6          | 23,823.2          |
| 2                 | 8,098.5           | 11,543.7          |
| 4                 | 4,734.7           | 5,306.8           |
| 8                 | 2,316.5           | 2,670.7           |
| 16                | 1,157.1           | 1,176.2           |
| 32                | 650.3             | 511.4             |
| 64                | 351.8             | 157.7             |

Table 11: LU phantom comparison (in millions)

| number of threads | original execution | improved execution |
|-------------------|--------------------|--------------------|
| 1                 | 16,829.6           | 25,472.8           |
| 2                 | 8,750.8            | 12,532.6           |
| 4                 | 5,301.0            | 6,262.1            |
| 8                 | 2,711.5            | 3,154.3            |
| 16                | 1,476.3            | 1,634.2            |
| 32                | 883.0              | 872.8              |
| 64                | 603.1              | 517.6              |

Table 12: LU execution time (in millions of seconds)

We believe our work compliments this work nicely in that we can provide the highly tuned applications to his dynamic scheduler.

Rafael Saavedra did much work characterizing various benchmarks [4] by decomposing them into high-level Fortran statements. Saavedra's work demonstrated that measurements of high level constructs can result in accurate predictions (over 50% of the predictions had less than 10% error) if cache or compiler optimizations are not used. In [5], he added terms in his model to account for cache effects. This improved the accuracy of his model to more than 80% of the predictions with less than 10% error. Our work compliments Saavedra's work in quantifying and understanding the interaction between kernels.

## 5 Summary

In this paper, we developed the coupling parameter to encompass multithreaded applications with two equations 1 and 2. These equations were then used to study three applications from the Splash-2 benchmark suite. The studies lead to improvements in LU and radix sort up to 55% decrease in phantoms and 14% decrease in execution time.

The coupling equations were developed to apply to other multiprocessor approaches (messagepassing and multi-paradigm) by adjusting the  $\mathcal{F}$  and  $\mathcal{E}$  functions from Equation 1 and Equation 2 appropriately. This allows a single function to be used across architectures and applications.

## References

- [1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *International Conference on Supercomputing*, June 1990.
- [2] Jonathan Geisler. Performance coupling: A methodology for analyzing application performance using kernel performance. Master's thesis, Northwestern University, March 1999.
- [3] Jonathan Geisler. *Performance Coupling for Sequential and Parallel Applications*. PhD thesis, Northwestern University, 2003.
- [4] Rafael H. Saavedra and Alan Jay Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report CSD-92-715, University of California, Berkeley, 1992.
- [5] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark run times. Technical Report CSD-93-767, University of California, Berkeley, 1993.
- [6] Allan Snaveley and Larry Carter. Symbiotic job-scheduling on the Tera MTA. In *Proceedings of Third Workshop on Multi-Threaded Execution, Architecture, and Compilers*, January 2000.
- [7] Allan Snaveley, Nick Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. Explorations in symbiosis on two multithreaded architectures. In *Proceedings of Second Workshop on Multi-Threaded Execution, Architecture, and Compilers*, January 1999.
- [8] Valerie Taylor, Xingfu Wu, Jonathan Geisler, and Rick Stevens. Using kernel couplings to predict parallel application performance. In *HPDC 2002*, July 2002.
- [9] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.