Locality Management in Task-Based Parallel Programming Models

Constantinos Koukos

Thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science at the:
University of Crete
School of Sciences and Engineering
Computer Science Department
Knossos Av. Heraklion, GR-71409, Greece

Thesis Advisors: Prof. D.S.Nikolopoulos, Prof. A.Bilas

Work performed at the:
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
Computer Architecture and VLSI Systems (CARV) Laboratory
100 N. Plastira Av. Vassiliki Vouton, Heraklion, GR-70013, Greece

Abstract: In this thesis we discuss the performance impact of cache locality on multicore x86 architectures. Our goal is to improve performance and efficiency on these systems. We suggest a novel task-based, parallel runtime system (TPR), that allows programmer to create custom scheduling policies and perform guided prefetching on the task arguments with minimal training effort requirements. TPR is targeted on very low overhead task creation, issue and synchronization. This system is designed to facilitate the programmer with utilities such as explicit data annotations to improve locality via prefetching and task binding to implement custom scheduling policies. Using that system we explore the parameters that affects locality in a major degree and the performance impact of each one. We conclude that the most important locality parameters are memory layout, task size, scheduling policy and SW prefetching. For the evaluation we use both linear algebra kernels and real applications. The kernels discussed are GEMM, LU, Cholesky, and Jacobi. The applications are PBPI, FixedGrid, and FFT. Finally we compare our work with other task based runtime systems and hand tuned applications written using pthreads or MPI. We manage an average performance improvement over other runtime systems from 1% to 40% depending on the application. The contribution of this work is to set a minimal subset of requirements for a runtime system to efficiently handle locality and to categorize applications and assign their primary optimization technique.
University of Crete  
Computer Science Department

Locality management in task-based parallel programming models

Thesis submitted by  
Constantinos Koukos

in partial fulfillment of the requirements for the  
Master of Science degree in Computer Science

THESIS APPROVAL

Author:  
Constantinos Koukos

Committee approvals:  
Dimitrios S. Nikolopoulos  
Associate Professor, Thesis Supervisor

Angelos Bilas  
Associate Professor

Manolis G.H. Katevenis  
Professor

Departmental approval:  
Panos Trahantias  
Professor, Director of Graduate Studies

Heraklion, August 2010
Περίληψη

Στην εργασία αυτή εξετάζουμε την επίδραση της τοπικότητας στην απόδοση των κρυφών μνημών για πολυπύρηνες αρχιτεκτονικές /DC/BK/BI/BA. Στόχος μας είναι η βελτίωση των επιδόσεων και της απόδοσης σε αυτά τα συστήματα. Προτείνουμε ένα πρωτότυπο, παράλληλο, σύστημα χρόνου εκτέλεσης που βασίζεται σε εργασίες (tasks), το TPR (Task Parallel Runtime), που στοχεύει σε μικρό χρόνο απόκρισης στην εκτέλεση των εργασιών και μικρή προσπάθεια από τον προγραμματιστή στην εκτέλεση των εργασιών. Το TPR επιτυγχάνει χαμηλό κόστος για την δημιουργία, χειρισμό και συγχρονισμό νέων εργασιών. Το σύστημα έχει σχεδιαστεί για να διευκολύνει τον προγραμματιστή και χρήστη να διευκολύνει την προανάκληση και την προσαρμοσμένη εκτέλεση των εργασιών. Το TPR έχει σχεδιαστεί για να διευκολύνει τον προγραμματιστή με λειτουργίες όπως την προανάκληση και την προσαρμοσμένη εκτέλεση εργασιών. Στην εργασία αυτή διερευνούμε τις παράμετρους που επηρεάζουν την τοπικότητα σε μεγάλο βαθμό και την επίδραση στην απόδοση της κάθε μιας. Καταλήγουμε στο συμπέρασμα ότι οι σημαντικότερες παράμετροι της τοπικότητας είναι η διάταξη της μνήμης, το μέγεθος των παράμετρων της εργασίας, η πολιτική προγραμματισμού και η βελτίωση της τοπικότητας. Για την αξιολόγηση χρήσιμοι τόσο μικροεφαρμογές (GEMM, LU, Cholesky και Jacobi) όσο και πραγματικές εφαρμογές (PBPI, FixedGrid και FFT). Τέλος, συγκρίνουμε την δουλειά μας με άλλα σύστημα χρόνου εκτέλεσης και δημιουργείμενες από τον προγραμματιστή εφαρμογές σε pthreads ή MPI. Καταφέρουμε μια μέση βελτίωση των επιδόσεων σε σχέση με άλλα συστήματα χρόνου εκτέλεσης από 1% έως 40%, ανάλογα με την εφαρμογή. Συνολικά, η συνεισφορά της εργασίας αυτής είναι να καθορίζει την βελτίωση των εφαρμογών ενός συστήματος χρόνου εκτέλεσης για να χειριστεί αποτελεσματικά την τοπικότητα, να αναθέσουμε τις εφαρμογές και να αναθέσουμε σε κάθε μια την βασικότερη βελτιστοποίηση.
# Contents

1 Introduction .......................... 1
  1.1 Motivation ........................ 1
  1.2 Profiling Methodology .......... 1
  1.3 Hardware Platform Specifications 2
  1.4 Related Work .......................... 2

2 Task Parallel Runtime (TPR) .......... 4
  2.1 TPR Interface .......................... 4
  2.2 TPR Architecture .................... 6

3 Locality Management .................. 9
  3.1 Memory Layout ......... 9
  3.2 Task Size and Layout ......... 10
  3.3 Scheduling Policy ........ 10
  3.4 Prefetching ........ 11

4 Methodology .......................... 14
  4.1 GEMM .......................... 14
  4.2 LU ................................ 17
  4.3 Cholesky ...................... 19
  4.4 Jacobi ................................ 21
  4.5 PBPI .......................... 21
  4.6 FixedGrid .......................... 23
  4.7 FFT .......................... 24

5 Evaluation .......................... 25
  5.1 Runtime .......................... 25
  5.2 GEMM .......................... 26
  5.3 LU .......................... 32
  5.4 Cholesky .......................... 34
  5.5 Jacobi - SOR .................. 35
  5.6 PBPI .......................... 37
  5.7 FixedGrid .......................... 38
  5.8 FFT .......................... 40
6 Comparison
   6.1 GEMM .................................................. 41
   6.2 LU ...................................................... 42
   6.3 Cholesky ............................................... 43
   6.4 FFT .................................................... 45
   6.5 PBPI ................................................... 46

7 Conclusions and Future Work 49

8 Acknowledgments 50
List of Figures

4.1 LU dependence graph ........................................ 18
4.2 Cholesky dependence graph .................................. 19
5.1 GEMM peak performance ...................................... 27
5.2 GEMM TLB Profiling .......................................... 27
5.3 GEMM Block Size Performance Impact ...................... 28
5.4 GEMM Block Size TLB Impact ............................... 28
5.5 LU peak performance ......................................... 30
5.6 LU TLB Profiling .............................................. 31
5.7 LU Block Size Performance Impact ......................... 31
5.8 LU Block Size TLB Impact ................................... 32
5.9 Cholesky peak performance .................................. 33
5.10 Cholesky TLB Profiling ..................................... 34
5.11 Jacobi peak performance .................................... 35
5.12 Jacobi TLB Profiling ........................................ 36
5.13 PBPI performance ............................................. 37
5.14 FixedGrid 3 species performance .......................... 39
5.15 FixedGrid 79 species performance ........................ 39
5.16 FFT peak performance ...................................... 40
6.1 GEMM TPR vs Cilk peak performance on 4D layout ...... 42
6.2 TPR and Cilk utilization for GEMM ......................... 43
6.3 LU TPR vs Cilk peak performance on 4D layout .......... 44
6.4 TPR and Cilk utilization for LU ............................ 44
6.5 Cholesky TPR vs Cilk peak performance on 4D layout .. 45
6.6 TPR and Cilk utilization for Cholesky ...................... 46
6.7 TPR vs Cilk vs SPLASH2 on FFT ........................... 47
6.8 TPR and Cilk utilization for FFT ............................ 47
6.9 PBPI on Cray-XT4 with 1 MPI process and 4 threads per node 48
## List of Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>GEMM locality analysis -RR-</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>GEMM locality analysis -Temporal-</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>RunTime Evaluation</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>GEMM cache profiling</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>LU cache profiling</td>
<td>33</td>
</tr>
<tr>
<td>5.4</td>
<td>Cholesky cache profiling</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>SOR cache profiling</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>PBPI cache profiling</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>TPR vs Cilk on GEMM</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>TPR vs Cilk on LU</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>TPR vs Cilk on Cholesky</td>
<td>43</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Last decade has been characterized by the hardware developers as the era of Terra. Multicore, multiprocessor systems are already capable of TerraFLOPs and is expected that in the near future a single CPU will be capable of such performance. The tradeoff to the performance gained by multicore processors is the programming complexity. x86 multicore systems are shared memory systems where each core has a private cache hierarchy and there is a higher level shared cache with an efficient cache coherent protocol. These memory hierarchies are not explicitly managed by the programmer. Instead these systems allow the programmer to give hints to the HW prefetcher about the data with the help of builtin prefetch instructions.

Locality management is one of the most important performance parameters. It is targeted to reduce data cache and TLB misses resulting to lower program execution time. The contribution of this work is to facilitate the programmer with the ability to improve locality on strided or irregular memory accesses by supplying an efficient, general purpose data prefetch mechanism. In addition to that the programmer is allowed to explicitly manage task assignment to a thread to create custom scheduling policies.

Finally the runtime is guaranteed to provide very low task creation and synchronization overheads leading to efficient execution on fine grain tasks. Fine grain tasks on a good scheduling policy are considered to optimize cache performance. Contrary to MPI or pthreads models our work is targeted to fine granularity and provides a simpler interface. We manage to find a minimal subset of requirements that allows the programmer to efficiently handle the most important locality parameters on a low development cost.

1.2 Profiling Methodology

The runtime has built in support for time and hardware profiling. Time profiling is achieved via the rdtsc register. This allow us to measure fine grain time slots with
an average latency of 23 core cycles. Hardware profiling information is gathered by using [1] (PAPI) over perfctr. The events monitored are L1 data cache accesses, L2 data cache accesses, L2 data cache misses and TLB misses. On systems that do not support all 4 events simultaneously statistical multiplexing is used by default. In our evaluation, both PAPI multiplexing and external multiplexing are used to verify the correctness of the results.

1.3 Hardware Platform Specifications

For the evaluation we use a 16 hardware thread system. Is based on 2 quad-core Intel Xeon E5520 processors. Each core has two active hardware threads over 8MT, sharing L1 and higher caches. Each core has a private L1 data cache of 32KB 4-way set associative and a private L2 of 256KB 8-way set associative while a shared L3 of 8MB is also available. The TLB has 64 Level-1 entries and 512 Level-2 entries. Processors communicate over Intel’s QPI (Quick Path Interconnect) capable of 5.86 GT/s. Each processor has 6GB of installed DDR3 triple channel RAM operating at 1066MHz. The processor additionally supports Intel Turbo Boost Technology that allows processor cores to run faster than the base operating frequency if it’s operating below power and temperature specification limits.

1.4 Related Work

The closest platform to the runtime developed is our previous work of TPC [2] for the IBM-CBE processor. TPC (Tagged Procedure Calls) is designed with similar semantics to TPR. A tagged procedure call is an asynchronous function call handled by the runtime. TPC task handling is transparent to the end programmer and does not support pinned tasks. TPC uses variable argument lists of 1 to 8 arguments for tasks and requires a task dispatcher written by the end programmer. Type casting of arguments and base address computation for each one is performed inside that dispatcher. TPR relieves the programmer from that responsibility by using an internal dispatcher and polymorphic functions. TPC needs each argument to be described as a triplet of \( \{\text{address, type, size}\} \) in a similar way with TPR. This is an architecture restriction that TPC has to comply. The only way to transfer data to the Cell processor SPE’s scratch memory is via explicit DMA transfers making that descriptor necessary.

We compare our work with Cilk-5 as described in [3]. Cilk-5, interface is very similar to TPR. To create a Cilk task the programmer has to annotate the task function with the keyword \texttt{cilk}. To issue a task the programmer annotates function call with the keyword \texttt{spawn}. Finally, Cilk provides a synchronization primitive called \texttt{sync}. Cilk programs requires a front end Cilk compiler to preprocess files. Task scheduling with Cilk is transparent to the end programmer. Cilk schedules tasks internally using work stealing on a cactus stack model.
Additionally we compare our work on large scale systems to StarSS [4]. For the comparison we use a native MPI application, PBPI [5] on a Cray-XT4 supercomputer. StarSS interface requires that the programmer annotates blocks of code or functions with \#pragma's to create a task. StarSS argument is that programmer should not express dependencies. The runtime instead performs a dynamic dependence analysis and dynamic scheduling of tasks. StarSS requires compiler and runtime support. The comparison was made with a version of StarSS that used hybrid MPI and task based runtime to overlap communication with computation as described in [6].

Sequoia [7] is another task based programming model that uses C/C++ language extensions. Sequoia allows the programmer to perform explicit memory management and is mainly targeted to distributed systems, such as the IBM-CBE processor. Tasks in Sequoia are scheduled using map reduce methods. OpenMP tasks [8], is another approach on task based parallel computing. This system use \#pragma annotations to create tasks and is mainly targeted to iteration based parallelism. OpenMP mainly provides synchronization primitives such as atomic, barrier and critical section and different scheduling policies such as static, dynamic or guided.

The advantages of TPR over the above runtime systems is that it facilitates the programmer with the ability to perform guided prefetching via annotations and allow explicit scheduling of tasks via pinned-tasks. In this work we further study the impact of different memory layouts referred to [9] and locality scheduling [10] via loop transformations. Loop transformations are compiler techniques to improve locality by creating a static schedule that has improved spatial or temporal locality over the initial. Such transformations have no runtime overhead and can be automatically generated for some classes of applications such as linear algebra kernels.
Chapter 2

Task Parallel Runtime (TPR)

The runtime concept is to create a system that keeps C semantics to ensure interoperability of programs and ease on porting existing applications. TPR does not reserve any keywords and can be compiled with any C compiler. No training effort is required by the programmer besides C or C++ experience. We have implemented both C and C++ interfaces for the runtime. The idea is that every C function with void * arguments and no return value can be executed asynchronously in parallel as a task. We define tasks as an asynchronous function call of a C function with a minimal of 1 and maximum of 8 arguments.

2.1 TPR Interface

To create a task the programmer must call: task (function_name, arguments). The C++ interface uses function overloading to alias all types of tasks to a single name. The use of variable arguments list (va_args) for task creation is highly discouraged because of the overhead it has and the limited ability of C compilers to perform static checking to the number and type of arguments. The C++ prototype for tasks with one and two arguments is shown in listing 2.1.

The interface is extended to a maximum of 8 arguments. These tasks are automatically scheduled by the runtime. To facilitate the programmer with the ability to create custom scheduling policies we provide pinned tasks. Pinned tasks are tasks explicitly bind to a thread. These tasks are guaranteed to follow FIFO execution on the specified thread. The interface of pinned tasks for one and two arguments is shown in listing 2.2.

The first argument (tQ) is the index of the thread that the task will be bind and must be within range \{0, \ldots, (\#MaxThreads-1)\}. Finally we provide the option to prefetch tasks data before execution by adding the type and size of the argument for both pinned or not tasks. This forms a triplet of: \{address, type, size\} for each argument. The interface of a task with prefetch hints for runtime and user scheduled tasks are shown in listings 2.3, 2.4 respectively.

Available options for type are: IN, OUT, INOUT voluntary masked by STRIDE.
CHAPTER 2. TASK PARALLEL RUNTIME (TPR)

Listing 2.1: Non-Pinned Task Interface

```c
void task(
    void (*func)(void *),
    void *a0
) ;
void task(
    void (*func)(void *, void *),
    void *a0, void *a1
) ;
```

Listing 2.2: Pinned Task Interface

```c
void task(
    uint16_t tQ,
    void (*func)(void *),
    void *a0
) ;
void task(
    uint16_t tQ,
    void (*func)(void *, void *),
    void *a0, void *a1
) ;
```

Stride arguments are those that use many rows of fixed size each with a fixed offset between them. These tasks are useful for prefetching 2D blocks on 2D arrays when memory is reserved using the C memory layout. We supply the macro `BUILD_STRIDE_ARG(#rows, row_size, offset)` for both programmer’s ease and internal packing reasons. The programmer must observe this contract even when he intends to prefetch a subset of the arguments. For that reason the option IGNORE_ARG is provided. When IGNORE_ARG is used as value to type or size field the argument will not be prefetched. Excluding arguments from prefetching is useful to apply selective prefetching.

Before task creation the runtime must be initialized using `init(#Threads)`. If 0 is used the runtime will detect and set the number of threads equal to the number of hardware threads. If 1 is used the runtime will run a sequential or C elision of the program executing each task upon creation on the main thread. Any other value \( N \) greater than 1 will lead to parallel execution using \( N \) threads. For synchronization we provide a global `barrier()` primitive. Example shown in listings 2.5, 2.6 illustrates a simple runtime use.

To parallelize the program we partition the dataset to the number of threads selected and assign a block to each task. The `add` function is transformed to polymorphic by changing the arguments type to `(void *)` and then cast back to
CHAPTER 2. TASK PARALLEL RUNTIME (TPR)

Listing 2.3: Non-Pinned Task Prefetch Interface

```c
void task (
    void (*func)(void *),
    void *addr, uint8_t type, uint64_t size
) ;

void task (
    void (*func)(void *, void *),
    void *a0, uint8_t t0, uint64_t s0,
    void *a1, uint8_t t1, uint64_t s1
) ;
```

Listing 2.4: Pinned Task Prefetch Interface

```c
void task (
    uint16_t tQ,
    void (*func)(void *),
    void *addr, uint8_t type, uint64_t size
) ;

void task (
    uint16_t tQ,
    void (*func)(void *, void *),
    void *a0, uint8_t t0, uint64_t s0,
    void *a1, uint8_t t1, uint64_t s1
) ;
```

the proper type. *term* terminates the runtime instance. No barrier is needed to that example because tasks are independent. *term()* guarantees that all tasks have finished before returning.

2.2 TPR Architecture

Asynchronous task execution is achieved by using internal runtime queues. Upon each task creation a descriptor containing all task information is created. The minimal information needed is the function address and the arguments addresses, while it can additionally store arguments size and type for prefetching. There are two types of queues, *private* and *shared*. Each thread including main thread has a private and a shared queue. Both types of queues are fixed size and NUMA aware. Pinned tasks use private queues only, because of their FIFO guarantees. Shared queues are used for load balancing via work stealing. Every thread can steal a task from any remote shared Q but our scheduling scheme ensures that this happens only when local private and shared queues are empty. Both types of queues use atomic synchronization primitives supported by the architecture. For private queues the
Listing 2.5: Sequential C example

```c
// Initial C written program
float A[256], B[256], R[256];
void add(float *a, float *b, float *r, int N) {
    int i;
    for (i = 0; i < N; i++)
        r[i] = a[i] + b[i];
}

int main ( void ) {
    add(A, B, R, 256);
    return (0);
}
```

The motivation is to synchronize one producer with one consumer. The implementation uses atomic increase/decrease primitives to the queue size. For shared queues there is one producer and many consumers. Synchronization is achieved with the use of bit scan forward and backward set/reset atomic primitives. Task descriptors are allocated to an internal allocator that uses atomic synchronization primitives and memory recycle. All queues and descriptors memory is allocated using NUMA aware techniques, and is small enough to fit in L1 cache.

Task issue is performed in a round robin manner for non-pinned tasks using both types of queues with respect to shared. If all threads queues are full then the task is executed immediately by the issuer (main) thread. In case of pinned tasks, the task can be issued only to the private Q of the thread selected. If selected Q is not full the task is inserted and the issuer returns. Otherwise the issuer will try to execute tasks from its own private Q or any shared Q, waiting for an empty slot in the selected remote Q. Work stealing is performed by workers when no more tasks exists in their local private or shared queues. Main and workers can also perform work stealing during a barrier.

When program reaches a barrier all tasks must be executed before returning. We use spinlocks with atomic primitives for barriers to reduce synchronization overhead. Atomic primitives used is architecture specific, bit test and set operations on a variable and an atomic zero operation. We implement a three step handshake protocol for barriers. Allowing work stealing on barriers impose use of the protocol twice creating two phases. Phase one allows work stealing overlapping idle time with computation. Because of the early update of barrier variables on worker side runtime may pass a barrier while some worker still processes a stolen task. Phase two ensure that such conditions will not occur. Phase two does not support work stealing because of the invariant that after phase 1 no more work stealing can be applied. The protocol involves three steps.


Listing 2.6: TPR parallel C example

```c
#include "tpr.h"

// Parallel with use of TPR
float A[256], B[256], R[256];
void add(void *a0, void *a1, void *a2, void *a3) {
    float *a = (float *)a0;
    float *b = (float *)a1;
    float *r = (float *)a2;
    int *N = (int *)a3;
    int i;

    for (i=0; i<*N; i++){
        *r[i] = *a[i] + *b[i];
    }
}

int main ( void ) {
    int N = 16;
    int i;

    init (16);
    for (i=0; i<256; i+=N){
        task(add, &A[i], &B[i], &R[i], &N);
    }
    term ();
    return (0);
}
```

1. The main thread atomically updates a flag to notify workers about barrier condition.

2. If a worker’s private and shared queues become empty upon barrier they atomically increase a shared variable and polls remote shared queues for work stealing.

3. The main polls shared variable until value = #workers. On the first phase of the barrier it tries to perform work stealing. When condition is satisfied then the main resets all flags and counters.

There are no point-to-point synchronization primitives supported. The reason is that pinned tasks can be used for RAW hazards elimination forcing the programmer to use temporal locality on such conditions.
Chapter 3

Locality Management

In this section we discuss the parameters that affect programs locality and the importance of each one. Modern shared memory multicore CPU’s are designed as a miniature of the past shared memory multiprocessor systems. The increase in the size of L1 and L2 caches does not follow the increase rate on the number of cores. To improve locality hardware developers constantly improve hardware prefetchers in these systems. Modern HW prefetchers can detect various memory access patterns of continuous or strided layouts or even prefetch data across pages. Prefetching data across pages is critical when using strided blocking kernels. Modern HW prefetchers operates on virtual addresses that makes them bound to TLB entries and TLB performance. We observe that an increase to the number of TLB misses results to a proportional increase in the cache miss ratio to all applications we tested. We conclude that locality is one of the most important parameters of performance. Although HW prefetchers can be very efficient on data prefetching there are several corner cases to be handled by a runtime system. We find in this thesis that in ascending importance order there are 4 parameters that improves locality

- Memory Layout
- Task Size and Layout
- Scheduling Policy
- Prefetching

3.1 Memory Layout

For a wide range of linear algebra kernels and real applications, techniques such as register tiling accomplished by SIMDization are still promising for good performance. Such techniques require blocking operations on data. In this work we explore two different memory layouts to apply blocking. Continuous or C memory layout in which memory is allocated within a sequential range of addresses for
single or multi-dimensional arrays. Tiled or 4D memory layout, in which for a 2D array we use a 2D array of pointers, each one keeping the base address of another 2D array of data.

Blocking in big or medium arrays using C memory layout can be seriously affected by TLB performance. Consider the case of a kernel operating on a 4Kx4K elements array with blocks of 32x32 single float elements. Each tile is exactly 4KB in both layouts. In the case of C memory layout strides will have an offset of 16KB (typically 4 pages) while in the case of 4D layout blocking results that memory is reserved to a continuous address range. If a task operates on such a tile in case of C memory layout TLB must have 32 entries reserved for this block while on 4D layout only 1 is needed. For blocking kernels or applications we suggest changing memory layout to 4D as the primary optimization. This is experimentally proved in section 5.

3.2 Task Size and Layout

Fine grain tasks have the advantage of exploiting more parallelism with the tradeoff of more runtime overhead while coarsening task size reduces overhead but has a negative impact to locality and can result to inefficient parallelism. For strided blocking tasks hardware prefetcher can fetch multiple sequential cache lines to improve spatial locality but there is a limitation to the number of simultaneous streams due to TLB restrictions. Best block sizes are those of mid granularity. For example, for all kernels we tested task sizes of 16x16 elements suffer from big runtime overhead and the fact that there are not enough data within each row to operate. Performance on block sizes greater than 128x128 is severely restricted by TLB misses and bad locality because of the HW prefetcher limitations. Typical block sizes that works well on all kernels are between 32x32 and 64x64.

3.3 Scheduling Policy

Scheduling policies try to improve locality by reordering tasks. Scheduling can decrease spatial or temporal distance depending on the application or kernel. We define spatial distance as the offset created from the higher address of predecessor task strides to the lower address of the successor task strides. When the second address range is sequential to that of the first we have zero spatial distance. When address ranges overlap we have partial or full temporal reuse. We define temporal distance as the number of bytes interpolated between two reuses on the same data. A simplest approach of temporal distance can use number of tasks instead of bytes.

As base scheduling policy for our experiments we use a round robin scheduler that uses work stealing. Depending on the kernel or application we can have scheduling policies that improves spatial or temporal locality. We use hand tuned spatial or temporal schedulers when applicable. Scheduling is managed by loop reordering and pinned tasks. Loop reordering involves dependence analysis of tasks
and in many cases requires changes in the dependence graph to avoid RAW hazards. Temporal scheduling is applicable to iterative kernels such as Jacobi and GEMM and iterative applications such as FixedGrid.

There is an invariant that if a temporal algorithm can be applied to some application using C memory layout and the application can support 4D layout then temporal scheduling is applicable to 4D layout as well. Spatial scheduling tries to schedule 2 tasks that use sequential address ranges in the same thread. In most of the cases this scheduling can be applied to applications or kernels using C memory layout. In some cases a pseudo-random scheduler can be used to avoid cache evictions due to address range collisions. An example of that is the GEMM kernel as shown in section 5. A benefit of 4D memory layout is that such conditions rarely occur due to heap randomization.

3.4 Prefetching

Prefetching is not a general purpose method to improve performance. It can be used as an important optimization for codes with strided or irregular memory accesses. Prefetching is affected by four main parameters:

- Accuracy
- Timeliness
- Distance
- Replace policy

Prefetching reduces task time by overlapping data transfers from main memory to cache with computation but has a small issue overhead. TPR time profiling, supply information about that overhead when prefetching is used. By running the application twice, with and without prefetching we measure: TaskTime_{original} which is task time without prefetch, TaskTime_{pfetch} which is task time when prefetch is used, and the prefetch overhead. We have a good use of prefetching when:

\[\text{TaskTime}_{\text{original}} - \text{TaskTime}_{\text{pfetch}} \geq \text{Overhead}_{pfetch}\]

TPR allow us to perform selective prefetching of arguments to reduce unnecessary prefetch issues. Prefetching accuracy is related to the scheduling policy. For instance, if temporal scheduling is applied to a kernel that operates on one block, then selective prefetching will try to prefetch data on the first task and avoid data prefetch to all other tasks that reuse the same data. If more blocks are used by the application and temporal locality scheduling is applied to some of them, then the previous is extended to selectively prefetch the arguments that are not reused.

In case of spatial scheduling selective prefetching is applied to the first task of the row or stride of rows.
CHAPTER 3. LOCALITY MANAGEMENT

Another parameter of prefetching is timeliness. Our runtime by default prefetches all arguments hinted for prefetch strictly before task execution. This decision was taken after extended testing to kernels and real applications. We observe that when prefetching is performed earlier it can cause cache evictions to the data of the task processed or the task processed can evict prefetched data. Many compilers use techniques such as prefetch loop arrays. These techniques can improve performance when accomplished by loop unrolling and the compiler can statically detect the data access pattern to the array. In this case the compiler spread some prefetch issues to the data of the next loop iteration. This method works well on regular data access patterns that a HW prefetcher can be very efficient as well. Our approach is that the user must hint the data to be prefetched and all data hinted will be in cache regardless of the access pattern before task execution.

The third parameter of prefetching refers to the distance of addresses that each prefetch instruction will start within block. We find that using a distance of quad cache lines (each cache line is considered to be 64Bytes) leads to the optimal tuning for all kernels and applications.

Prefetching instructions are machine specific and depends on the implementation of the processor. There are no guarantees about when a prefetch instruction will be issued and there is no common interface across manufacturers. For instance Intel prefetch instructions can be followed by a hint about locality with four different modes while AMD provides only one mode. These hints can be used to set the replace policy of the cache. AMD prefetches data to L1 cache while Intel uses higher levels. To prefetch data we use gcc builtin_prefetch with the higher locality as default. Machine specific demands are handled by gcc.

In most cases we gain from prefetching when no scheduling is applied or when using C memory layout on strided arguments. Prefetching techniques described earlier are applicable to task sizes that can fit in cache and TPR guarantees that such fine grain tasks have a small runtime overhead. Example in listing 3.1 shows how selective prefetching can be applied for a GEMM algorithm with temporal scheduling. Matrix size used is N and the number of HW threads is P. Temporal scheduling is applied to the result matrix R and for that reason no prefetching is performed to the blocks of R matrix. The example uses C memory layout. Matrices memory is statically reserved using 2D arrays of MAX_MATRIX_SIZE each.
void GEMM_temporal ( void ) {
    uint32_t i, j, k, ti, pi;
    uint32_t offset = MAX_MATRIX_SIZE*sizeof(float));
    uint32_t row_size = BlockSize*sizeof(float));

    for (i = 0; i<N; i+-(BlockSize *P)) {
        for (j = 0; j<N; j+=BlockSize) {
            for (k=0; k<N; k+=BlockSize) {
                for (ti=0; ti<P; ti++) {
                    pi = i+(ti*BlockSize);
                    task(
                        ti, block_multiply_transposed,
                        &R[pi][j],1,IGNORE_ARG,IGNORE_ARG,
                        &A[pi][k],IN_ARG | STRIDE_ARG),
                        BUILD_STRIDE_ARG(BlockSize,row_size,offset),
                        &B[j][k],IN_ARG | STRIDE_ARG),
                        BUILD_STRIDE_ARG(BlockSize,row_size,offset),
                        &BlockSize,IGNORE_ARG,IGNORE_ARG
                    );
                }
            }
        }
    }
    barrier();
}
Chapter 4

Methodology

In this section we further explain techniques to improve locality on linear algebra kernels and applications. Not all methods mentioned above are applicable to every kernel. Kernels used in this work are GEMM, LU, Cholesky and Jacobi. Applications are FFT, PBPI, and FixedGrid.

4.1 GEMM

GEMM kernels performs matrix multiplication with subtraction or addition operations. Consider 3 matrices A, B and R where A, B are read-only containing the input data and R is read-write used as the result matrix with in place data update. Kernel performs \( R = R + (A \times B) \) or \( R = R - (A \times B) \) operations to the matrices. We implement a blocking square GEMM kernel by using SIMD SSE instructions and register tiling of 4x4 elements. For good cache performance the minimum block used is 16x16 elements. Processor has L1 cache line size of 64Bytes so such blocks typically use 16 cache lines. Additionally all blocks starting addresses is guaranteed to be 64 Bytes aligned. Before GEMM we perform a blocking in-place transpose of B matrix to avoid column-wise element accesses. Algorithm 1 shows issuing of GEMM tasks with internal round robin scheduling. The algorithm can be applied to C memory layout with step of BlockSize. Kernel can support 4D memory layout by converting matrices to 4D layout using a step of one and limit iterations to MatrixSize /BlockSize.

In this scheduling scheme, from now on called “RR”, loop carrying reuse is aliased to A tiles. Maximum temporal reuse is: \( \frac{\text{MatrixSize}}{\text{BlockSize} \times \# \text{Threads}} \). In this scheme there are \( \left( \frac{\text{MatrixSize}}{\text{BlockSize}} \right)^2 \) independent tasks across barriers. Table 4.1 analyzes spatial and temporal reuse for all matrices blocks. For table 4.1 “Temporal” refers to temporal reuse and is the number of sequential tasks operating on the same block. “Spatial” refers to spatial distance and is the number of tasks interleaving until proceed to the successor block. If a block is reused then this value is computed using the last access on the block. Finally “T.R.D.” stands for temporal reuse distance and is the number of tasks interleaving between two sequential accesses.
Table 4.1: GEMM locality analysis -RR-

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Temporal</td>
<td>Spatial</td>
<td>Temporal</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>(N/\text{Block} (j))</td>
</tr>
<tr>
<td>((N/\text{Block})^2)</td>
<td>0</td>
<td></td>
<td>(N/\text{Block})</td>
</tr>
</tbody>
</table>

Table 4.2: GEMM locality analysis -Temporal-

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Temporal</td>
<td>Spatial</td>
<td>Temporal</td>
</tr>
<tr>
<td></td>
<td>(N/\text{Block} (k))</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>(N/\text{Block})</td>
<td>((N/\text{Block})^2)</td>
<td></td>
</tr>
</tbody>
</table>

to the same block.

The second scheduling policy we applied, tries to bind locality to R tiles in a manner that a final result is produced for each R tile and the next tile to proceed is the row-wise successor of the current. To use this algorithm all intermediate steps that produces a final result on any R tile must comply to FIFO ordering due to RAW hazards. We use pinned tasks for this algorithm, from now on called “Temporal” as shown in alg. 2.

“Temporal” algorithm has the good ability to have \(\frac{\text{MatrixSize}}{\text{BlockSize}}\) tasks on the same R tile, reused by the same thread. Notice that temporal reuse is independent to \#Threads. Another feature of this algorithm is that it tries to keep max spatial locality on R matrix by binding groups of tasks row-wise on the same thread. The inner \(ti\) aliased loop, performs load balancing ensuring that one task is issued to every thread for each step of computation. Notice that in this policy only 1 barrier is needed while previous algorithm needs \(\frac{\text{MatrixSize}}{\text{BlockSize}}\) barriers. Table 4.2 presents locality behaviour of the algorithm.

The last algorithm used for this kernel is the “Random” algorithm in which during initialization we create an index to store consistent \(\{i, j\}\)\(\forall k\) for all A, B, R tiles. Scheduling is performed in an RR fashion as shown in alg. 3.

The idea of this algorithm is to show how can scattered blocking memory accesses affect performance and how efficient can HW prefetcher be in such accesses. Summarizing for GEMM we evaluate the impact of different memory layouts, scheduling policies, block sizes, and prefetching.
Algorithm 1 Blocking GEMM RR Scheduling

for $k := 0$ to MatrixSize by BlockSize do
  for $i := 0$ to MatrixSize by BlockSize do
    for $j := 0$ to MatrixSize by BlockSize do
      task (GEMM, &R[i][j], &A[i][k], &B[j][k]);
    end for
  end for
end for

Algorithm 2 Blocking GEMM Temporal Scheduling

for $i := 0$ to MatrixSize by (BlockSize*#Threads) do
  for $j := 0$ to MatrixSize by BlockSize do
    for $k := 0$ to MatrixSize by BlockSize do
      for $ti := 0$ to #Threads by 1 do
        $pi = i+(ti*BlockSize)$;
        task($ti$, GEMM, &R[$pi$][j], &A[$pi$][k], &B[j][k]);
      end for
    end for
  end for
end for

Algorithm 3 Blocking GEMM Random Scheduling

for $i := 0$ to (MatrixSize/BlockSize) by 1 do
  for $j := 0$ to (MatrixSize/BlockSize) by 1 do
    for $k := 0$ to (MatrixSize/BlockSize) by 1 do
      task (GEMM, &R[$i*BlockSize$][j*BlockSize],
            &A[$i*BlockSize$][Index[k][i][j]*BlockSize],
            &B[j*BlockSize][Index[k][i][j]*BlockSize]);
    end for
  end for
end for

barrier();
4.2 LU

We implement a single float, blocking, dense LU kernel similar to the one used in [11]. Our approach is more convenient for task based environments. SPLASH-2 LU kernel or other hand tuned LU kernels implement functions such as BlockOwner or DiagonalOwner to assign affinity of code sections into threads using conditional statements. Such techniques finally lead to more complicated code. If such codes are directly translated to task based environments will result in creating tasks whose execution will depend on internal conditions which is against good design of task based applications. We prefer a straightforward design approach in which task scheduling is performed outside task and all conditional checks referring task creation are performed before task spawning. The sequential LU algorithm used is shown in alg. 4. Blocking transformation of this kernel involves splitting the kernel into four sections.

1. Diagonal Factorization
2. Column-wise Factorization
3. Row-wise Factorization
4. GEMM

We used the SIMDized GEMM kernel from previous section while the three first kernels are performing scalar operations. The amount of time spent on the three scalar kernels is 11.8% while the rest 88.2% performs SIMD operations. Kernel \{2,3\} blocks are dependent on kernel 1 blocks. Blocks operating GEMM have dependencies on blocks of \{2,3\} kernels as shown in figure 4.1.

The simplest schedule for LU is the “RR” described in alg. 5. It involves three barriers. The first barrier ensure that steps \{2,3\} are processed after finishing step 1. The second barrier eliminates hazards across steps \{2,3\} and 4, while the last one guarantees that all computation on current step have finished before proceed. Algorithm is presented in alg. 5.

Due to dependencies mentioned above is far more complicated to create a schedule that takes advantage of temporal locality because across LU steps active dataset size is reduced. Instead we implement a scheduling policy that tries to keep good spatial locality in GEMM tasks by using pinned tasks. The algorithm is similar to RR with the exception that GEMM tasks are split into row-wise groups and each group is assigned to a thread. Kernel operates in way that number of tasks are reduced on each pass. This can create imbalances when the right lower part of the matrix is to be processed. We statically improve the schedule to use non-pinned tasks and work stealing when number of remaining strides equals to the number of threads. Spatial scheduling cannot be applied on the kernel when using 4D memory layout. Summarizing for LU we evaluate the impact of two memory layouts (C and 4D), two scheduling policies for C memory layout, different block sizes and prefetching for both layouts.
Algorithm 4 LU Base Algorithm

```
for i := 0 to MatrixSize by 1 do
    for j := i + 1 to MatrixSize by 1 do
        A[j][j] := A[i][i];
        for k := i + 1 to MatrixSize by 1 do
            A[j][k] := A[j][i]*A[i][k];
        end for
    end for
end for
```

Algorithm 5 Blocking LU RR Scheduling

```
for i := 0 to MatrixSize by BlockSize do
    task ( factorizeDiagonal, &A[i][i] );
    barrier();
    for j := i + BlockSize to MatrixSize by BlockSize do
        task ( factorizeRow, &A[i][i], &A[i][j] );
    end for
    barrier();
    for j := i + BlockSize to MatrixSize by BlockSize do
        for k := i + BlockSize to MatrixSize by BlockSize do
            task ( GEMM, &A[j][i], &A[j][k], &A[i][k] );
        end for
    end for
    barrier();
end for
```
CHAPTER 4. METHODOLOGY

4.3 Cholesky

We implement a single float, blocking, dense Cholesky decomposition kernel similar to the one mentioned in [12]. Cholesky decomposition can only be applied to symmetric, positive definite matrices. The algorithm operates on lower triangular part of the matrix. The algorithm is shown in alg. 6 and can be decomposed into 4 main functions

1. Diagonal Factorization
2. Column-wise Factorization
3. Triangular GEMM Factorization
4. Square GEMM Factorization

As with LU we use only square GEMM SIMD kernel and leave the rest kernels unvectorized. This results to a time distribution of 10.3% scalar and 89.7% SIMD operations. Because Cholesky operates only to the lower triangular part of the matrix each GEMM block depends on the row-wise block and the anti-diagonal column-wise one. This forms a more complicated dependence graph than LU as shown in figure 4.2.

Because of the imbalances Cholesky forms we applied only an “RR” scheduling scheme described in alg. 7 for both memory layouts. As mention earlier to create a temporal scheduling we must have an iterative kernel that reuse the same data. Cholesky processes in a way that task number is reduced on each step of the algorithm. To apply spatial scheduling we usually need a kernel that is in C memory layout and task affinity can be set in a way that different tasks takes advantage of sequential addresses. Cholesky decomposition is triangular and such a technique will lead to imbalances with a severe performance penalty. For Cholesky in addition to memory layout we evaluate different block sizes, and prefetching.
Algorithm 6 Cholesky Base Algorithm

for \( k := 0 \) to MatrixSize by 1 do
    \( A[k][k] = \sqrt{A[k][k]}; \)
    for \( j := k + 1 \) to MatrixSize by 1 do
        \( A[j][k] /= A[k][k]; \)
    end for
    for \( j := k + 1 \) to MatrixSize by 1 do
        for \( i := j \) to MatrixSize by 1 do
            \( A[i][j] -= A[i][k] * A[j][k]; \)
        end for
    end for
end for

Algorithm 7 Blocking Cholesky RR Scheduling

for \( i := 0 \) to MatrixSize by BlockSize do
    task ( factorizeDiagonal, &A[i][i] );
    barrier();
    for \( j := i + \text{BlockSize} \) to MatrixSize by BlockSize do
        task ( factorizeCol, &A[i][i], &A[j][i] );
    end for
    barrier();
    for \( j := i + \text{BlockSize} \) to MatrixSize by BlockSize do
        task ( triangularGEMM, &A[j][i], &A[j][j] );
        for \( k := j + \text{BlockSize} \) to MatrixSize by BlockSize do
            task ( GEMM, &A[k][i], &A[k][j], &A[j][i] );
        end for
    end for
    barrier();
end for
4.4  Jacobi

There are two basic Jacobi algorithms in the literature. For more information please refer to [12]. The Jacobi method (4-point) stencil and the SOR (or Gauss-Seidel) method (5-point stencil). The first method requires two matrices. One is used as read only and the other one as write only. After each iteration the two matrices are swapped. This algorithm tasks are independent because no updates are performed in place but have the disadvantage of using double the actual memory needed, leading to bad locality. The mathematical formulation of this algorithm is:

\[ x_i^{(k+1)} = \left( b_i - \sum_{i \neq j} a_{ij} x_j^{(k)} \right)/a_{ii}, \ i = 1, \ldots, N \]

The SOR algorithm operates in place so only one matrix is needed resulting to improved locality but suffers from all to all communication that leads to low parallelism. It can be written as:

\[ x_i^{(k+1)} = \left( b_i - \sum_{i < j} a_{ij} x_j^{(k+1)} - \sum_{i > j} a_{ij} x_j^{(k)} \right)/a_{ii} \]

To combine good locality with higher parallelism we implement a hybrid method of asynchronous or chaotic relaxation. It uses SOR kernel within each block reading updated values for the inner elements and previous iteration values for the border elements. This algorithm have a slower convergence rate than SOR but it can be efficiently parallelized. The borders of each block are forming a grid that is updated across iterations in parallel using memory copies.

SOR kernel is fully vectorized including aligned and unaligned SIMD load instructions. All arithmetic operations performed are SIMD. This algorithm can be scheduled using both round robin “RR” and “Temporal” policies. Temporal scheduling assigns each block to a processor forming a stride of tasks in a row-wise manner to keep good spatial locality additional to temporal. The algorithm is shown in alg. 8. This algorithm can be applied to both memory layouts. In case of 4D memory layout it will only take advantage of temporal locality while in C memory layout it will additionally improve spatial locality.

In this kernel we used a fixed iterations threshold “T” for evaluation ease. Additionally to that we pre-cache an index array of the starting row index for each thread. If the matrix is small enough to fit in cache one could expect that all cache misses will happen in the first iteration and there will be no evictions during program lifetime. Evaluation gives supporting evidence on this assumption, concluding that the most important optimization for iterative kernels is temporal scheduling. Upon a bigger dataset performance will depend on the cache replace policy, HW prefetcher and spatial locality.

4.5  PBPI

PBPI, Parallel Bayesian Phylogenetic Inference is fully described in [5]. PBPI is a parallel implementation of Bayesian phylogenetic inference method for DNA sequence data. It’s an improvement of the bayesian phylogenetic inferences with Markov Chain Monte Carlo (MCMC) method with likelihood-based assessment of
Algorithm 8 Blocking SOR Temporal Scheduling

\begin{verbatim}
for k := 0 to T by 1 do
  for i := 0 to RowsPerThread by 1 do
    for j := 0 to #Blocks by 1 do
      for ti := 0 to #Threads by 1 do
        task ( ti, GridUpdate, &Grid[Index[ti]+i][j] );
      end for
    end for
  end for
  barrier();
end for

for i := 0 to RowsPerThread by 1 do
  for j := 0 to #Blocks by 1 do
    for ti := 0 to #Threads by 1 do
      task ( ti, SOR, &Grid[Index[ti]+i][j] );
    end for
  end for
  barrier();
end for
\end{verbatim}

phylogenies. PBPI is integrated with MPI to perform many levels of parallelism such as:

- Dataset level parallelism
- Run level parallelism
- Partition level parallelism
- Chain level parallelism
- Sequence level parallelism

On each node there are four 1D arrays that the kernel operates. Operations mainly involve additions and multiplications to quads of elements and are performed on three loops. The final loop is responsible to compute the likelihood of the node. We apply SIMDization over SSE2 and loop based iteration parallelism for all loops. Because the dataset consists of 1D arrays memory layout transformations are not applicable. Additionally no prefetching is used because the memory accesses of the application are very regular.

The original MPI version is spatial locality aware. Trying to statically schedule a temporal scheme is not feasible because each node processed is decided by the likelihood computed on each step. We apply a finest level of parallelism at dataset level over MPI schedule. In this way we are able to directly compare our runtime with coarse grain hand tuned MPI parallelism. We implement two scheduling
CHAPTER 4. METHODOLOGY

schemes. An “RR” that span spatial locality across cores and a “Spatial” schedule similar to native. The second does not use pinned tasks. Instead it partitions dataset # into # Threads sections where each section contains fine grain tasks and issues tasks round robin across sections. Because runtime uses round robin scheduler final placement of tasks will be equivalent to spatial scheduling. Each thread Q will have tasks on a sequential address range. With this technique we can perform spatial scheduling with work stealing enabled on independent tasks.

4.6 FixedGrid

FixedGrid is a chemical transport model for mass balance equations solving. Such equations are formed from traces of air species determining atmosphere pollution. For more information about the chemical model please refer to [14]. FixedGrid can perform operations on various species. By default processes ozone concentration while dataset used involves from 3 to 79 species. FixedGrid dataset consists of 5 matrices. There is one 3D matrix to store concatenation. The Z dimension is time while X,Y represents geographic coverage. Operations involving this matrix named “con” are performed in place. The other 4 matrices are 2 dimensional and contains information about wind and diffusion tensors and their transposed ones. FixedGrid performs 3 main operations with the same kernel. It performs two row-wise discretizations with an intermediate column-wise one. The column-wise discretization can be replaced with a matrix transpose and row-wise discretization for better cache locality. Wind and diffusion tensors are read only so we built a transposed replicate of them during initialization.

For FixedGrid we implement 2 different scheduling policies “RR” and “Temporal” on 2 different memory layouts, C and 4D. For the C layout transposes on the “con” matrix are performed across discretize steps while 4D packs transposes to column-wise discretize tasks. For the C memory layout discretize operates on a matrix row and all discretize tasks for one step are independent. By the way the kernel operates to elements there are dependencies from each element to the two left and right neighbours in case of row-wise discretization or to the two upper and bottom when performing column-wise discretization. The 4D layout forms a grid similar to that of Jacobi using SOR method. Grid consists of blocks with the actual data and a separate replicate for each border. Border elements are updated across algorithm steps (3 times on each pass). This technique eliminates dependencies by using a few memory copies. Transposes are performed for this layout before and after block discretization and are packed on the same task to improve locality. “Temporal” scheduler uses an index created at startup to assign affinity to a group of tasks that reuses the same data. FixedGrid is not amenable to prefetching because memory accesses are very regular. Although we tested all cases using prefetch we could not manage any performance improvement. For that reason it is not included in the evaluation.
4.7 FFT

We use a port of [11], six step FFT algorithm with blocking transpose as described in [13]. Briefly, this algorithm performs three transposes and two 1-Dimensional FFT’s between them. FFT’s are performed row-wise for cache spatial efficiency. Although the algorithm touches all elements within a row these accesses forms a butterfly to the powers of 2 that can be considered as irregular even for the most sophisticated HW prefetchers. A typical state-of-the-art HW prefetcher has limited lookahead because of branches, buffer size and irregular array indexing, while long cache lines have false sharing. FFT operations involves a triple loop that dynamically changes number of iterations and array access pattern. The algorithm perform the following steps:

1. Transpose $N_1 \times N_2 \rightarrow N_2 \times N_1$
2. Perform $N_1$ individual $N_2$ FFT’s
3. Multiply $A_{ij}$ by $e^{\pm 2\pi ijk/n}$
4. Transpose $N_2 \times N_1 \rightarrow N_1 \times N_2$
5. Perform $N_2$ individual $N_1$ FFT’s
6. Transpose $N_1 \times N_2 \rightarrow N_2 \times N_1$

Because of the way FFT’s touches data within a row 4D memory layout is not applicable. Task size can be tuned only for blocking transpose operations while FFT’s use a single row per task. For a matrix of N elements there are $\sqrt{N}$ independent tasks of $\sqrt{N}$ elements each, during FFT1D operations Each element is a complex double with sequential reserved memory for the real and imaginary parts. We use only one scheduling scheme trying to use temporal locality across the six steps of the algorithm and spatial on each step.

The primary optimization for this application is prefetching. Prefetching is applied to transposes because they have strided memory accesses and to FFT1D operations because they have irregular memory access pattern. Compared to SPLASH-2 FFT version in our implementation all operations are performed in place to the main FFT matrix while SPLASH-2 copy elements during transpose using double the memory we use and resulting to lower cache performance in big datasets.
Chapter 5

Evaluation

In this section we perform an evaluation of the techniques used for each kernel and application. We show the impact of each parameter that affects locality. For the graphs we make a convention to use the following acronyms. “C-ML” means C memory layout, “4D-ML” is 4D memory layout. For scheduling policies we use “RR” for round robin scheduling, “Temp” for temporal scheduling etc. Prefetching is marked as “PF” when used or not mentioned when not used. For example a label 4D-ML 128x128 Temp PF mean the run is using 4D memory layout on 128x128 block tiles accomplished by temporal scheduling and prefetching.

5.1 Runtime

Runtime system developed targets low overhead task creation, issue, and completion. We manage to implement a runtime system with an average latency of 400 core cycles per task, issue to completion overhead. Runtime evaluation is performed by using a simple NULL task test that executes tasks without data and empty statement body. We perform a coarse grain measurement of one million tasks spread across all threads including main. This test use both private and shared queues and work stealing. During this test all runtime statistics are disabled because of the overhead they put on such fine grain measurements. The rdtsc latency is about 23 core cycles and at least four polls to the register are required, totalling 100 core cycles overhead to measure both issuer and worker overhead. Instead, we use gprof and adjust the percentage to the real time measured by rdtsc. gprof instrumentation increases total time by about 5% and uses the same compiler optimization parameters as the runtime. Results are taken using 4 cores on single processor and are shown in Table 5.1.

Total time measured can be split three main parts. Completion time which is the time spent at each worker without including task time. Completion step involves a lookup at the workers private and shared queues respectively. If a task is found on any local queue the worker dequeue the task and execute it. After task execution the worker deallocates task descriptor memory. If local queues are
empty worker constantly tries to steal a task from any remote shared queue. When no tasks exist in any queues local or remote then polling time to queues can be characterized as idle time. Issue time is the time spent on the main thread to create a task descriptor and assign it to a thread. Main thread polls all shared and private queues in a round robin manner once. If all queues are full it executes the task immediately. Otherwise main allocates a task descriptor and enqueues the task to the queue earlier detected and return to user code. Finally time spent to execute the NULL function is characterized as task time. We observe that the time spent to issue is nearly 35% while completion phase takes 55% of the time. This is explained because the rate one main thread can produce tasks is lower than the rate four workers can consume tasks. On real applications completion time which includes rdtsc overhead, dequeue time, dealloc time, and idle time is measured to less than 180 core cycles.

Overheads shown in Table 5.1 are measured on real applications giving us extra confidence for the runtime overhead. A 5% increase to issue time is noticed when issuing tasks of chip while the completion time remains intact for each worker due to NUMA allocation used on queues and task allocators. It is very difficult to accurately evaluate barrier overhead because of work stealing that overlaps idle time and the imbalances caused by each application. We find with the use of micro-benchmarks that each phase of the barrier protocol discussed in section 2 have an average overhead of nearly 2800 core cycles on 16 HW threads using off chip communication and SMT. Replacing atomic primitives on the variable with pthread mutexes we measure an overhead of 56K core cycles on the same setup. Using native pthread barrier have an overhead of 50K core cycles.

### 5.2 GEMM

GEMM is one of the most computationally intensive kernels. It can be used as a standalone kernel and is embedded in LU and Cholesky kernels as well. The GEMM version we implement performs blocking operations with register tiling and is fully vectorized. GEMM performance can be seriously affected by memory layout. Peak performance is gained by using 4D memory layout as shown in Fig. 5.1. GFLOPS are computed as: $\text{GFLOPS} = \frac{(2 \times N^3)}{\text{Time}}$. In this case study runtime overhead was less than 0.85%. All scheduling policies for the 4D memory layout have a small

<table>
<thead>
<tr>
<th>#Arguments</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>399</td>
<td>396</td>
<td>444</td>
<td>525</td>
<td>100.00%</td>
</tr>
<tr>
<td>Issue</td>
<td>137.256</td>
<td>136.224</td>
<td>152.736</td>
<td>180.6</td>
<td>34.40%</td>
</tr>
<tr>
<td>Complete</td>
<td>222.642</td>
<td>220.908</td>
<td>247.792</td>
<td>292.95</td>
<td>55.80%</td>
</tr>
<tr>
<td>Task</td>
<td>39.102</td>
<td>38.808</td>
<td>43.512</td>
<td>51.45</td>
<td>9.80%</td>
</tr>
</tbody>
</table>
Figure 5.1: GEMM peak performance

Figure 5.2: GEMM TLB Profiling
CHAPTER 5. EVALUATION

Figure 5.3: GEMM Block Size Performance Impact

Figure 5.4: GEMM Block Size TLB Impact
CHAPTER 5. EVALUATION

difference near 2%. The best scheduling is Temporal followed by RR and finally Random. By using 4D layout we gain at least 12.33% on performance compared to the best C memory layout schedule. Graph on fig. 5.1 uses the best block size for both layouts which is 128x128. During evaluation we tested different block sizes.

Block sizes that are not multiple of 4 elements cannot be fully vectorized. Block sizes that are not multiple of 16 elements (64Bytes) causes cache lines false sharing and accesses that are cache line unaligned with a severe performance penalty. This restricts block sizes with good performance to a subset of those that are 16 elements multiple. We observe that block sizes of 96x96 or 144x144 have reduced performance compared to 128x128. That leads to the conclusion that best block size for GEMM is 128x128 elements. For the rest we use some more representative block sizes.

For the C memory layout Temporal locality schedule is the best. This schedule has 6% improvement over the next better schedule which is Random. Using HW profiling we find that Random schedule miss ratio is very close to temporal schedule on that block size. This is because HW prefetcher can efficiently handle streams on that granularity. Using smaller block sizes Random scheduling have reduced cache efficiency. For block sizes of 128x128 the difference in performance is caused by TLB misses. Random schedule causes 33.8% more TLB misses than Temporal on the selected setup. RR scheduling performance is affected by cache data conflicts. This makes Temporal scheduling 2.5 times faster than RR.

Performance improvement of 4D memory layout over C layout comes from TLB misses as shown in fig. 5.2. In general 4D layout causes 1 order of magnitude less TLB misses than C memory layout. Block size can seriously affect performance as shown in figure 5.3. Results are taken using the best scheduling policy and memory layout. We observe that GEMM performs well on block sizes of 128x128 and 64x64 elements. Block size also affect TLB misses as shown in fig. 5.4. That explains why coarse block sizes have improved performance. Further increasing block size reduces TLB performance. For the results on this graph we use 4D memory layout with temporal scheduling. Using Random scheduling and doubling block size results to a decrease of more than 1 order of magnitude to TLB misses.

On GEMM prefetching is noticed to improve performance in cases where we use C memory layout with medium block sizes (ex. 32x32) and RR or Random scheduling applied. Maximum performance gain measured from prefetching was about 9% while maximum negative impact was about 10%. We notice a negative impact of prefetching when using 4D memory layout and temporal scheduling. Selective prefetching can reduce the overhead in the latter but cannot further improve performance. Table 5.2 shows cache performance for the setup shown in figure 5.1 for 16 HW threads.
Table 5.2: GEMM cache profiling

<table>
<thead>
<tr>
<th>%MissRatio</th>
<th>L1</th>
<th>L2-Local</th>
<th>L2-Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>C RR</td>
<td>11.021</td>
<td>47.158</td>
<td>5.197</td>
</tr>
<tr>
<td>C temp</td>
<td>7.268</td>
<td>56.436</td>
<td>4.101</td>
</tr>
<tr>
<td>C rand</td>
<td>7.114</td>
<td>57.628</td>
<td>4.099</td>
</tr>
<tr>
<td>4D RR</td>
<td>6.385</td>
<td>48.298</td>
<td>3.084</td>
</tr>
<tr>
<td>4D temp</td>
<td>4.521</td>
<td>48.114</td>
<td>2.175</td>
</tr>
<tr>
<td>4D rand</td>
<td>6.394</td>
<td>48.663</td>
<td>3.111</td>
</tr>
</tbody>
</table>

Figure 5.5: LU peak performance
CHAPTER 5. EVALUATION

Figure 5.6: LU TLB Profiling

Figure 5.7: LU Block Size Performance Impact
5.3 LU

LU performance is improved by memory layout transformation with 4D memory layout being 66.7% faster than C memory layout. For the 4D layout of LU no scheduling policies could be implemented as explained in section 4.2 while C memory layout supports RR and Spatial scheduling. For the C memory layout we find that both policies manage similar performance with RR slightly better because of work stealing. The HW prefetcher is capable of keeping good spatial locality on both policies. Additionally to that C memory layout gains about 2% from prefetching on both policies while 4D layout gets a negative impact to performance around 1%. On LU we apply all arguments prefetching for RR schedule and selective prefetching on Spatial schedule. Fig. 5.5 sums the total view. GFLOPs for LU are computed using the following approximation formula: 

\[ GFLOPS = \frac{(2/3) \times N^3}{Time}. \]

For LU the best block size is 64x64 with a little difference of about 2% from 32x32. Block sizes of 16x16 puts great runtime overhead while block sizes of 128x128 reduce granularity and leads to imbalances as shown in fig. 5.7. LU using 4D memory layout causes an order of magnitude less TLB misses than C memory layout as shown in fig. 5.6 while task size affects TLB by almost 1 order of magnitude as well. Impact of block size to TLB performance is shown in fig. 5.8. The spike in fig. 5.6 on the C memory layout from 8 to 16 threads is caused
by the use of SMT. Using SMT, 2 threads on the same core share the same TLB so for the block size used we have evictions due to structural reasons. Table 5.3 shows cache performance. LU RR scheduling policy on C memory layout has lower cache miss ratio than Spatial. That explains the performance difference shown in figure 5.5. Spatial scheduling policy tries to improve cache performance by optimizing spatial locality. For LU, GEMM step operates on three blocks on the same matrix. Tasks are synchronized in a way that for all tasks processing different blocks, the same column-wise block (from step 2) is used by all tasks. This create replicates to the private caches and finally leads to lower cache performance than RR.

Table 5.3: LU cache profiling

<table>
<thead>
<tr>
<th>%MissRatio</th>
<th>L1</th>
<th>L2-Local</th>
<th>L2-Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-ML RR</td>
<td>9.225</td>
<td>82.925</td>
<td>7.649</td>
</tr>
<tr>
<td>C-ML Spatial</td>
<td>11.495</td>
<td>88.990</td>
<td>10.229</td>
</tr>
<tr>
<td>4D-ML</td>
<td>2.053</td>
<td>6.974</td>
<td>0.143</td>
</tr>
</tbody>
</table>

Figure 5.9: Cholesky peak performance
5.4 Cholesky

Cholesky kernel performs blocking operations on a non-symmetric way forming an irregular dependence graph. As explained in section 4.3 we could not perform different scheduling policies. Best block size for Cholesky is 32x32 followed by 64x64. Block size of 32x32 combines maximum granularity on a low runtime overhead. Performance improvement of block size between 2 best block sizes is about 14.8% as shown in fig. 5.9. Cholesky gains 38.8% speedup by using 4D memory layout. We find that prefetching improves Cholesky performance on both memory layouts and all block sizes at a maximum of 6%. Because no scheduling policy can be applied we apply full prefetching to all task arguments. Prefetching benefit from 8 to 16 HW threads on both memory layouts leads to the conclusion that prefetching is an important optimization for Cholesky to reach top performance.

4D layout causes two orders of magnitude less TLB misses than C memory layout as shown in fig. 5.10. Block size can affect TLB misses by nearly one order of magnitude. Despite the unbalanced behaviour of Cholesky work stealing manages to reduce runtime overhead to nearly 5% while another 2% is spent on prefetching resulting to top performance. GFLOPs are computed using: \( GFLOPS = \frac{(1/3) \times N^3}{Time} \) formula. Cache performance for Cholesky is shown in table 5.4.
5.5 Jacobi - SOR

SOR is very sensitive to block size and scheduling policy. The best block size is 128x128 elements while bigger or smaller blocks have a severe performance impact. Similar with GEMM we exclude 64Bytes unaligned blocks. As mentioned in section 4.4 the algorithm we implement use memory copies to update grid. Smaller block sizes puts unnecessary overhead because of more memory copies. Coarsen block sizes could not be tested because of the dataset size used. GFLOPs are computed using: \( GFLOPS = \frac{6 \times N^2 \times T}{Time} \) where \( N \) is the matrix size and \( T \) is the number of iterations until convergence.

For simplicity reasons we use a fixed threshold till convergence. We observe that Jacobi performance is negative affected by SMT because of TLB and cache evictions due to structural reasons and has been excluded from the results. Despite other kernels Jacobi is more intensive to scheduling policy than to memory layout.
Figure 5.12: Jacobi TLB Profiling

Temporal scheduling is 29.3% faster than RR scheduling while memory layout affects performance only by 7.7% when Temporal scheduling is performed. If RR scheduling is applied memory layout affects performance by 22.72%. As seen in fig. 5.11 the best memory layout is 4D.

Jacobi gains nearly 1% speedup from prefetching when C memory layout is used on a RR schedule. When 4D memory layout is used prefetching reduces performance because of the overhead it puts. When temporal scheduling is applied selective prefetching is the best option because of the low overhead but does not manage any performance improvement. Prefetching blocks on each task on temporal schedule puts great overhead without reducing task time and should be avoided. We also observe that TLB misses are affected by 2 orders of magnitude by tempo-

<table>
<thead>
<tr>
<th>%MissRatio</th>
<th>L1</th>
<th>L2-Local</th>
<th>L2-Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-ML RR</td>
<td>7.097</td>
<td>89.091</td>
<td>6.323</td>
</tr>
<tr>
<td>C-ML Temp</td>
<td>5.924</td>
<td>88.655</td>
<td>5.252</td>
</tr>
<tr>
<td>4D-ML RR</td>
<td>9.536</td>
<td>53.347</td>
<td>5.087</td>
</tr>
<tr>
<td>4D-ML Temp</td>
<td>10.038</td>
<td>48.099</td>
<td>4.828</td>
</tr>
</tbody>
</table>
CHAPTER 5. EVALUATION

PBPI Nexus L-50000 Double Time

![PBPI Nexus L-50000 Double Time Graph](image)

Figure 5.13: PBPI performance

...racial scheduling and other 2 by memory layout as shown in fig. 5.12. This leads to the conclusion that SOR is more sensitive to Temporal scheduling than memory layout improvement. Table 5.5 shows SOR cache performance for different memory layouts and scheduling policies without prefetching.

5.6 PBPI

PBPI base version is written in MPI and supports many levels of parallelism as mentioned in previous section. We implement a dataset level parallelism with 2 scheduling policies, Spatial and RR. The dataset used was the largest available NEXUS-50000L. We use a maximum generation of 10000 with a sample interval of 100. We use a single Markov Chain for testing which involves MPI and runtime barriers and is one of the hardest tests of PBPI. All kernels used are fully vectorized over SSE2 instruction set.

In fig.5.13 we can see that Spatial policy improves performance by 20% over RR. The original MPI version scale very well and is near our best version with the exception of using 16 HW threads over SMT. In case of multithreading both our versions scales better than the original because of the finest granularity they have and the lower synchronization overhead of the runtime. In case of TPR with RR scheduling using SMT we observe the following phenomenon. Threads sharing the...
Table 5.6: PBPI cache profiling

<table>
<thead>
<tr>
<th>%MissRatio</th>
<th>L1</th>
<th>L2-Local</th>
<th>L2-Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPR RR</td>
<td>6.460</td>
<td>66.684</td>
<td>4.308</td>
</tr>
<tr>
<td>TPR Spatial</td>
<td>6.094</td>
<td>63.437</td>
<td>3.866</td>
</tr>
</tbody>
</table>

same core, shares L1, L2 caches and TLB and the affinity of this threads results to sequential numbering. Using RR scheduling assigns two neighbor blocks to the same core resulting to less TLB misses and better cache efficiency. That is the reason for the speedup from 8 to 16 threads using RR. Notice that although we have a performance improvement we do not reach the peak managed by Spatial scheduling. This is because after the second task on the same core we have a strided access for the next couple of tasks. Spatial scheduler on the other hand requires HW prefetcher to use double the streams needed without SMT but leads to a stride of sequential blocks per core. HW prefetcher is efficient on handling that condition keeping the same performance with 8 HW threads. We could not expect some further improvement of the application using SMT for structural reasons explained above.

Finally we observe that the use of SMT for native version leads to reduced performance. The reason is the expensive interprocess communication used for double number of threads that cannot give a further speedup due to structural reasons. Table 5.6 illustrates the difference in cache miss ratio for 8 HW threads on different scheduling policies.

5.7 FixedGrid

Default implementation of FixedGrid uses C memory layout. Kernel operates on a row-wise fashion. By default FixedGrid is spatial locality aware. We implement two different distributions of the application, one using C memory layout and another using 4D. For each one we develop 2 different scheduling policies, Temporal and RR. For big datasets of 1024x1024 elements scheduling does not affect performance as shown in figures 5.14 and 5.15. Improvement comes from the use of 4D memory layout and is 12.3% to 50% (depending on the number of cores) for 3 species and nearly 14% for 79 species. Temporal scheduling doesn’t improve performance because the dataset is too big to fit in cache.

Smaller datasets can gain about 9.7% from temporal policy when 3 species used but have the drawback that they need very fine grain tasks. Such fine grain tasks leads to poor scalability. Additionally when many (ex. 79) species are used even on a small geographic area, dataset becomes too big to fit in cache. That leads to the conclusion that although fixedgrid uses an iterative kernel realistic datasets can’t gain performance from temporal locality so the primary optimization should
Figure 5.14: FixedGrid 3 species performance

Figure 5.15: FixedGrid 79 species performance
be the use of 4D memory layout.

5.8 FFT

Six step FFT algorithm described in previous section has irregular memory accesses within task and forms dependencies that does not allow to change memory layout or further improve locality with scheduling. The algorithm discussed tries to benefit from both temporal and spatial locality. Locality is mainly affected by matrix size. Our test is performed to a quite big matrix of $2^{26}$ complex double elements (1 GByte). GFLOPs are computed according to the following formula: 

$$GFLOPs = \frac{(10 \times M \times N)}{Time}$$

where $N$ is the number of complex elements and $M = \sqrt{N}$. Performance is plotted to fig. 5.16. We notice an important performance improvement of prefetching of 3.45% while vectorizing the kernel resulted to a less than 1% improvement.
Chapter 6

Comparison

In this section we compare TPR with Cilk. Cilk is optimized for expressing recursive algorithms. For the comparison we use a port of our kernels to Cilk. The memory layout we use is the 4D and the scheduling policy is RR because Cilk does not provide an option to pin tasks. Additionally, in this section we compare our work on a distributed system using a hybrid implementation of TPR and MPI on PBPI.

6.1 GEMM

For GEMM, Cilk performs best and comparably to TPR by using 128x128 block size. For smaller tasks, we observe big differences in performance. Fig. 6.1 shows the impact of block size for Cilk and TPR. There is a small performance gain of TPR over Cilk by 2.7% using 128x128 elements blocks. Table 6.1 shows TPR speedup over Cilk for 8 and 16 HW threads. Fig. 6.2 shows utilization on both runtimes for the setup shown in Fig. 6.1. We define utilization as the percent of average task time for all threads including main. The rest is runtime overhead and idle time.

<table>
<thead>
<tr>
<th>BlockSize</th>
<th>16x16</th>
<th>32x32</th>
<th>64x64</th>
<th>128x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Threads</td>
<td>23.690</td>
<td>8.798</td>
<td>1.530</td>
<td>1.106</td>
</tr>
<tr>
<td>16 Threads</td>
<td>26.402</td>
<td>10.928</td>
<td>1.472</td>
<td>1.027</td>
</tr>
</tbody>
</table>
Figure 6.1: GEMM TPR vs Cilk peak performance on 4D layout

Table 6.2: TPR vs Cilk on LU

<table>
<thead>
<tr>
<th>BlockSize</th>
<th>16x16</th>
<th>32x32</th>
<th>64x64</th>
<th>128x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Threads</td>
<td>27.527</td>
<td>5.750</td>
<td>1.402</td>
<td>1.679</td>
</tr>
<tr>
<td>16 Threads</td>
<td>32.782</td>
<td>8.615</td>
<td>1.323</td>
<td>1.160</td>
</tr>
</tbody>
</table>

6.2 LU

Cilk achieves peak performance for LU with block size of 64x64 while TPR gets peak on 32x32 block sizes. Smaller tasks caused big overheads while bigger tasks lead to imbalanced performance. Comparing peak performance on both runtimes TPR is 35.22% faster than Cilk. Fig. 6.3 shows the impact of block size for Cilk and TPR. Table 6.2 shows TPR speedup over Cilk for 8 and 16 HW threads. Fig. 6.4 compare runtimes utilization for LU.
Figure 6.2: TPR and Cilk utilization for GEMM

Table 6.3: TPR vs Cilk on Cholesky

<table>
<thead>
<tr>
<th>BlockSize</th>
<th>16x16</th>
<th>32x32</th>
<th>64x64</th>
<th>128x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Threads</td>
<td>29.527</td>
<td>5.186</td>
<td>1.289</td>
<td>1.422</td>
</tr>
<tr>
<td>16 Threads</td>
<td>27.363</td>
<td>9.359</td>
<td>1.226</td>
<td>1.022</td>
</tr>
</tbody>
</table>

6.3 Cholesky

Cholesky on TPR achieves peak performance on fine grain tasks of 32x32 elements. Cilk has a big overhead using that block size and this affects the peak performance it can achieve. Cilk manages top performance on 64x64 elements blocks. TPR peak performance is 39.99% better than Cilk's peak performance. Fig. 6.5 shows the impact of block size for both runtimes. Table 6.3 shows TPR speedup over Cilk for 8 and 16 HW threads. Fig. 6.6 compare runtimes utilization for Cholesky.
CHAPTER 6. COMPARISON

LU 4096*4096 Single Float GFLOPs Peak

![Graph showing LU 4096*4096 Single Float GFLOPs Peak](image)

Figure 6.3: LU TPR vs Cilk peak performance on 4D layout

LU 4096*4096 Single Float Utilization

![Graph showing LU 4096*4096 Single Float Utilization](image)

Figure 6.4: TPR and Cilk utilization for LU
6.4 FFT

As earlier discussed FFT is an application with irregular memory accesses inside task. We observe that TPR is much more efficient (nearly 58\%) than coarse grain parallelism with pthreads as shown in fig. 6.7. This is mainly because SPLASH-2 FFT uses staggered, blocked copy transposes while TPR and Cilk kernel use square, blocking in-place transposes. Additionally TPR is 29.57\% faster than Cilk in case of 16 HW threads. For the Cilk version of FFT we use exactly the same kernel as TPR with the same configuration. We can see that performance difference increases proportional to the number of threads. This is caused by different barrier implementations.

We also observe that Cilk FFT scales in a similar way with SPLASH-2 FFT but has a constant performance increase of nearly 22\%. This is caused by the different transpose implementations mentioned earlier. Although SPLASH-2 transpose method is targeted for temporal locality on small FFT’s the dataset used is too big to fit in cache. This technique finally leads to use double the actual memory needed and bad spatial locality due to column-wise accesses increasing transpose fraction from 1/3 in task environments to 1/2 of the total computation time. Fig. 6.8 compare runtimes utilization for FFT.
CHAPTER 6. COMPARISON

6.5 PBPI

In this section we compare TPR with an x86 version of [4] (StarSS). Fig. 6.9 shows that there is great performance advantage of TPR over StarSS. The reason is the very low overhead of TPR that almost reaches the speedup of the native version. Maximum speedup on 64 cores is 55x using native version while TPR gets a 52x speedup. StarSS argument is that the runtime can schedule fine grain tasks over MPI to overlap computation with MPI communication on synchronization conditions. The runtime automatically detect dependencies and schedule independent tasks transparently. We measure that the analysis and scheduling overhead of the runtime is greater than time gain by overlapping, leading to lower performance than TPR or native application.
CHAPTER 6. COMPARISON

**Figure 6.7:** TPR vs Cilk vs SPLASH2 on FFT

**Figure 6.8:** TPR and Cilk utilization for FFT
Figure 6.9: PBPI on Cray-XT4 with 1 MPI process and 4 threads per node
Chapter 7

Conclusions and Future Work

In this work we present a runtime system with the minimal interface required to write high performance parallel applications. We show that different categories of programs have different requirements on scheduling. Blocking kernels such as GEMM, LU and Cholesky mainly benefit from a 4D memory layout. Using a temporal scheduling policy, if applicable, will further increase performance. For iterative kernels such as SOR the main optimization should be to implement a temporal scheduling policy. If dataset doesn't fit in cache, iterative kernels can benefit from 4D memory layout or spatial scheduling.

Non-regular memory access applications, such as FFT, which typically requires enormous effort to become blocking can get significant performance improvement from prefetching. For applications with very regular memory accesses and good spatial locality such as PBPI if no temporal scheduling can be implemented the programmer should expect good scalability as long as spatial locality is maintained.

We are strongly convinced that a low latency task based runtime system can be more or equally efficient than using threads or MPI model and much easier in use. Compared to other task based runtime systems our work has lower overheads leading to better utilization on fine grain tasks and facilitates the programmer with more advanced options such as prefetching or custom scheduling that can significantly improve locality with low development cost.

Techniques, such as loop reordering, have been implemented for most of the kernels resulting to a respectable performance improvement. Such techniques can be automated and applied statically by a compiler in order to help programmer. The idea of pinned tasks is that a compiler could produce such tasks to improve scheduling. Additionally, we are thinking of methods that could statically automate the memory layout transformation from C memory layout to 4D.

The TPR interface can support distributed architectures if the programmer annotates all task arguments. Creating an implementation of the runtime that can use hybrid parallelism of tasks on both architectures under the same interface is one of our targets. Finally, is under exploration whether the runtime should be expanded to support dynamic dependence analysis and dynamic locality scheduling.
Chapter 8

Acknowledgments

I would like to thank all the people who made this work possible. First, I would like to thank my supervisors Prof. Dimitrios S. Nikolopoulos and Prof. Angelos Bilas for their assistance and guidance during this work. Second, I would like to thank Prof. Manolis G.H. Katevenis for his guidance at the technical writing of this work. I would like to thank Michail Alvanos and Dr. Polyvios Pratikakis for their assistance during this work. I would like to extend my gratitude to all the people of the Computer Architecture and VLSI Systems Laboratory (CARV) of FORTH-ICS.

The work reported in this thesis has been conducted at the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH), and has been financially supported by a scholarship provided by FORTH-ICS.
Bibliography


