Multicore ZPL

By

Steven P. Smith

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2007

Presentation of work given on March 28, 2007
Thesis and presentation approved by __________________________
Date ______________________
Introduction

Over the last several years, the performance of computers has been improving exponentially as predicted by Moore’s Law. In recent years, the performance increase has come primarily in the form of additional processing engines, or cores, on each CPU instead of increases in raw CPU clock rate. While the overall power of these new multi-core CPUs is greater than that offered by any previous generation of processor, not all applications can take advantage of it; many applications were written and are still written for sequential execution. Sequential programs cannot take advantage of the new power of modern processors, and some sequential programs even suffer since the clock speed of each individual core has decreased slightly.

Although the circumstances are new, this problem is an old one; similar issues arise when even the most powerful single CPU is not fast enough for a particular job. In this case, computer designers are forced to either create a computer that contains multiple physical CPUs, or link multiple computers together via a network to generate the throughput the job requires. In order to utilize the performance gains these configurations provide, the software must be able to split its task into subtasks that can be executed in parallel; this allows as many CPUs as possible to execute a subtask, or thread, providing the best possible throughput. Designing multithreaded software can be challenging, however, and this difficulty may cause some programmers to favor sequential programs since they are easier to write, sacrificing run time to save coding time.

One tool that has been developed to make authoring multi-threaded programs easier is the ZPL programming language. ZPL is an array-based language in which the compiler handles all aspects of parallelism, so the programmer does not have to know how to divide certain kinds of tasks into multiple threads and get them all running together. While ZPL programs run faster than single-threaded programs on computers with multiple processors or on clusters of networked computers, there is overhead associated with running in these configurations that isn’t required on modern, multi-core CPUs; the aim of this paper is to detail the work that went into developing a mechanism to allow communication between concurrently running threads in a ZPL program with the goal of minimizing overhead.
ZPL
ZPL is a high-level parallel programming language developed at the University of Washington. It is based on a compiler that translates ZPL programs to C code with calls to MPI, PVM, or SHMEM, as the user chooses [1]. At a high level, ZPL is an array-based programming language that assumes the task of adding parallelism to a program. While ZPL is designed to use parallelism as much as possible, ZPL programs are sequential, with no user-written communications code; ZPL handles all aspects of parallelizing a program, allowing the programmer to focus on the problem at hand without having to worry about communications, synchronization, or any of the other issues that make parallel programming difficult. This allows the programmer to concentrate on correctly implementing the computation correctly, leaving the details to the compiler. The compiler will identify opportunities for parallelism, perform all inter-processor communication, implement synchronization where required, and perform extensive parallel as well as scalar optimizations. While there is much to be said about ZPL, others have already covered the language itself in much detail [1, 2, 3]. For the purposes of this paper, a quick overview of ZPL and a brief explanation of two of ZPL’s features should be sufficient.

Basic ZPL
ZPL offers many of the same features that other programming languages offer, including similar data types, operators, and control structures. ZPL offers boolean, float, double, quad, complex, signed integer, unsigned integer, char, and a few other types. It also offers the usual operators, including +, -, *, /, and !. Common control structures are also available, including if statements, while statements, and for statements. Although ZPL programs take advantage of parallelism whenever possible, ZPL programs are written sequentially, just like programs in other languages; the compiler handles the details of parallelism.

Regions
Regions are one of ZPL’s key concepts. A region is simply an index set over which arrays are declared and manipulated. Regions can have any number of dimensions and any bounds, and are declared like this:

region R = [1..n, 1..n];

The above statement declares a two-dimensional array R where each dimension varies from 1 to n. Arrays can be created over this region, and such arrays will have two dimensions covering the integer space from 1 to n.

Reductions
Reductions are another important concept in ZPL. Reductions allow the programmer to “reduce” the size of an array by combining its elements. Some of the more commonly used reductions are +<< (sum), *<< (product), &<< (bitwise AND), |<< (bitwise OR), max<< (determines the maximum value), and min<< (determines the minimum value).

The usefulness of reductions is best demonstrated by an example. Consider a two-dimensional integer array x declared on the region R defined above. To find the sum of all numbers in the array using a
language like C or Java, nested for loops are frequently used; nested for loops are inefficient on computers with multiple processors as only one processor can be used, and miscalculating the boundaries of each loop is a common mistake programmers make. Reductions make this computation very easy, requiring only one line of code:

\[
\text{sum} := +<< x;
\]

This computes the sum of all elements of array \( x \) and stores it in the variable \( \text{sum} \). Using a reduction for this computation eliminates the boundary calculations necessary when using for loops, and allows ZPL to parallelize the operation as much as possible to take advantage of multiple processors.

**Examples**

To better illustrate how ZPL programs work and how ZPL provides parallelism, consider the following two examples. The first is Conway's Game of Life, and the second is the Jacobi iteration, which models heat transfer.

**Game of Life**

Conway's game of life is a simple algorithm that is very straightforward to write in ZPL and benefits from the parallelism that ZPL offers. The game involves an area that is split into a grid. Organisms are initially placed into the grid, and the game starts. For each cycle of the game, an organism is born in a grid space if there are organisms in three neighboring spaces; an organism dies if it has more than 4 or fewer than 2 neighbors. Figures 1 – 3 show three cycles of the game.

![Fig. 1. Initial state of the Game of Life.](image1)

![Fig. 2. One cycle into the Game of Life.](image2)
ZPL Code
This is the code for the Game of Life algorithm, written in ZPL:

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
direction NW = [-1,-1];  N = [-1, 0];  NE = [-1, 1];
               W = [ 0,-1];  E = [ 0, 1];
               SW = [ 1,-1]; S = [ 1, 0]; SE = [ 1, 1];
var  NN : [R] ubyte;  TW : [R] boolean;
procedure Life();
    [R] begin
        /* Read in the data */
        repeat
            NN :=  TW@^NW + TW@^N  + TW@^NE
                  + TW@^W           + TW@^E
                  + TW@^SW + TW@^S  + TW@^SE;
            TW := (NN=2 & TW) | (NN=3);
        until ! |<<TW;
    end;
```

The world is represented by the array TW. Each element of TW indicates whether or not there is an organism living in that grid space. Each cycle around the loop, which ends when there are no more organisms in the world, the number of neighbors of each grid space is counted and this count is stored in NN; NN is another array that is the same size as TW, but holds the number of neighboring spaces of each space that are occupied by an organism. This operation happens for every space on the grid in parallel, taking advantage of the fact that the same operation is being applied to every element of an array. The birth and death algorithm is applied to TW based on the number of neighbors for each space, NN; this also happens in parallel, as the algorithm is applied to each element of TW based on the
number of neighboring organisms stored in the corresponding element of NN. Figures 4 – 8 show how this works through two cycles of the game.

![Fig. 4. Initial state of the Game of Life, stored in TW.](image1)

![Fig. 5. The number of neighbors of each grid space have been calculated and stored in NN.](image2)

![Fig. 6. The state of the Game of Life after one cycle. Each grid space has been updated based on the number of neighbors it had.](image3)

![Fig. 7. The number of neighbors of each grid space for the first cycle.](image4)

![Fig. 8. The state of the Game of Life after two cycles.](image5)

This program is very easy to implement in ZPL; the entire algorithm takes 4 lines of code (the example uses 6 for clarity). Note that looping to scan the arrays is not necessary in ZPL – arrays can be manipulated as if they were scalars. The compiler takes care of parallelizing the operations, so each update to an array uses as many processors simultaneously as possible; this makes the ZPL implementation both very efficient and easy to write.

**Jacobi**

ZPL can be very useful for solving problems that lend themselves to representation using arrays. One example is the Jacobi iteration, which models heat diffusing through a plate. The basic algorithm can be thought of as having four steps – initialize, compute new averages, update the array, loop until convergence. This is exactly the sort of problem that ZPL was designed to solve, so writing ZPL code to solve this problem is very easy; only one line of ZPL code is needed for each of the conceptual steps mentioned above. Figures 9 – 11 illustrate how the algorithm works.
Fig. 9. Initial state for the Jacobi iteration. The heat in the plate is represented by the elements with value 1 in the bottom row.

Fig. 10. After one iteration, the heat has started to spread. The value of each array element is the average of its four neighbors (above, below, left, right). The error in the computation is 0.12, so another iteration is necessary.

Fig. 11. After two iterations, the heat has spread further.
**ZPL Code**

The ZPL code required to implement the Jacobi algorithm is very straightforward.

```
program Jacobi;
config var n : integer = 6;
    eps : float = 0.00001;

region     R = [1..n, 1..n];
    BigR = [0..n+1,0..n+1];
direction  N = [-1, 0];  S = [ 1, 0];
    E = [ 0, 1];  W = [ 0,-1];
var        T : [R] float;
    A : [BigR] float;
    err : float;

procedure Jacobi();
    [R] begin
    [BigR] A := 0.0;
    [S of R] A := 1.0;
    repeat
        T := (A@N + A@E + A@S + A@W)/4.0;
        err  := max<< abs(T - A);
        A    := T;
        until err < eps;
    end;
end;
```

The above code implements the Jacobi iteration in ZPL. Note that only one line of code is necessary to compute the average of each element’s neighbors, and the computation of the maximum error is also computed using only one line of ZPL code. This highlights the ease with which array-based algorithms can be implemented using ZPL. In addition, ZPL will parallelize each line of code, handling all communication and synchronization between processors; this happens automatically so programmers don’t need to think about the complexities of parallel programming.

**Parallelism in ZPL**

ZPL handles parallelism by dividing work up among all available processors. When a ZPL program is executed, it divides up the available processors into a grid based on the number of dimensions that arrays in the program use; this results in a two-dimensional grid of processors for both of the previously mentioned examples. When arrays are allocated, each processor is given an approximately equal chunk of the array over which it will operate. Each processor will execute each statement that manipulates an array over the chunk of the array it was allocated, and the results of the computations from all processors are combined into the resultant array of the operation. As an example, consider a square array of integers and a statement that adds 1 to each element of the array. The sequential way to compute the result is to iterate over the rows and columns of the array, adding 1 to each element. ZPL
optimizes this by spreading the sequential work among many processors. If this example were part of a ZPL program running with two processors, each processor would be assigned half of the array to work on, so the result would be computed nearly twice as fast as the purely sequential way.

Communication in ZPL

One of the difficulties that arises in situations like this is what to do if a computation requires data that isn’t available. As an example, consider a program like Jacobi that averages the values of each element’s neighbors. If the program is running with two processors in a one column, two row arrangement, the processor responsible for the “top” half of the array will be able to run until it hits the last row in its half of the array. At this point, the values of the array one row down are held by the other processor; the “top” processor will need to ask the “bottom” processor to share the values necessary for the “top” processor to complete successfully.

In order for the “bottom” processor to share data it holds, it uses ZPL’s communication layer. The communication layer provides an interface that the compiler uses to allow processes to communicate; ZPL provides a series of implementations of this interface, called Ironman, which use different lower-level mechanisms to implement the communication functionality defined by the interface. The Ironman implementation is included in compiled ZPL programs, so the binary contains the low-level communication implementation. At this time, ZPL comes with Ironman implementations for sequential programs (no communication), MPI, PVM, and SHMEM.

Ironman

The specifics of this communication are handled by the Ironman layer. The current version of the ZPL compiler ships with Ironman implementations for MPI, PVM, and SHMEM. MPI is one of the more common message passing libraries used, and it can be used successfully on single computers with multiple processors. The problem with doing this is that MPI involves overhead necessary for multiple computers connected by a network; that overhead is not necessary on single computers with multiple processors, and reduces the speed at which ZPL programs can run. The goal of this research was to design an Ironman implementation that allows processors in a single computer to communicate without unnecessary overhead, which should yield the most efficient ZPL programs possible.
My Implementation

I chose to use POSIX IPC (Inter-Process Communication) for my Ironman implementation. IPC provides shared memory regions which multiple processes on a single computer can connect to, and shared semaphores, which are used to control access to shared resources. IPC doesn’t require the overhead of libraries like MPI, uses facilities that are built into many operating systems, and allows physical memory to be shared directly between multiple processes. In the case of multi-core processors, shared memory can take advantage of cache sharing between the cores, providing an additional increase in speed over multiprocessor computers. Although there are potentially more efficient ways to solve this problem, like the use of a thread library like pthreads, the API provided by IPC is very similar to that of the shared memory library used by the Cray T3E which was used in the SHMEM Ironman implementation; this allowed me to replace all calls to the shared memory library with equivalent calls to IPC.

To complete this task, I had to find all calls to the T3E’s SHMEM library, determine their behavior, and find comparable functionality provided by IPC. The SHMEM library provided some functionality that isn’t implemented by IPC, which I had to implement. As an example, I implemented a barrier function to synchronize processes across multiple processors since this functionality was not provided by IPC. To properly test my code, I wrote test programs that isolated my code from the rest of the Ironman code, as well as simple ZPL programs that involved only the language features I implemented.

Results

I was unable to complete enough of my implementation to benchmark it, though another student in the department was able to get an Ironman implementation for multi-core computers working using a different design. Nathan Kuchta also used IPC shared memory, but he limited his design to two processors while I did not place such limits on mine. He was able to get his working, so I performed some benchmarks on his implementation as an estimate of the performance I might see from my implementation. All benchmarks were conducted on two computers using two test programs. The results of 16 executions of each benchmark were collected, and the best execution time was used in all
calculations. Execution times from the sequential versions of both programs were used as a base execution time for calculating speedup.

**Computers**
The first computer was attu1.cs.washington.edu, which was equipped with four Intel Xeon 2.8GHz processors and four gigabytes of RAM, running a custom 2.6.20 Linux kernel. Since this computer is a shared machine, the benchmarks were run at several times during the day to ensure that load did not impact the scores. The second computer was x64, which was equipped with a single 2.2GHz AMD Athlon64 X2 4200+ dual-core processor and three gigabytes of RAM, running Fedora Core 6 in single-user mode.

**Benchmarks**
The first benchmark was red_blue (Appendix A), a program that runs a mathematical computation for a given number of iterations. The default value of 10000 was used. The other benchmark was a modified version of the Jacobi iteration. This version used three dimensions instead of two, did not compute the error at the end of each iteration, and used a problem space of 100 units over 100 iterations.
**Charts**

![Chart 13](image1.png)

**Fig. 13.** This chart shows the time, in seconds, required to run the `red_blue` (Appendix A) benchmark on `attu1`. The speedup gained by the multi-core Ironman implementation is 1.5.

![Chart 14](image2.png)

**Fig. 14.** This chart shows the time, in seconds, required to run the modified Jacobi program on `attu1`. The speedup gained by the multi-core Ironman implementation is 1.22.
Fig. 15. This chart shows the time, in seconds, required to run the red_blue (Appendix A) benchmark on x64. The speedup gained by the multi-core Ironman implementation is 1.58, which is slightly better than the 1.5 achieved on attu1.

Fig. 16. This chart shows the time, in seconds, required to run the modified Jacobi program on x64. The speedup gained by the multi-core Ironman implementation is 1.39, again slightly better than on attu1.
Limitations
My Ironman implementation has several limitations that need to be addressed. First, the implementation is incomplete; the semantics of the shared memory calls used by the SHMEM implementation don’t exactly match those of IPC, sometimes leading to unexpected behavior. I decided to limit the scope of my implementation to @-communication, leaving all of ZPL’s other features for future work. While this implementation strategy shows promise, using threads may be a better choice in the quest for an Ironman implementation with as little overhead as possible since threads simply share the address space of their parent process.

Conclusion
Computers are getting faster every year, though as of late this increase in performance has come in the form of additional processing cores. Writing software that makes use of more than a single thread of execution is a difficult task, and many programmers avoid the problems of parallel programming by writing sequential programs. To encourage programmers to write parallel programs that will run well on today’s multi-core computers, tools must be developed to make the task of writing a parallel program easier. ZPL accomplishes this goal well, allowing programmers to concentrate on implementing high-level logic while the compiler takes care of parallelism. ZPL’s current methods for communication between processors involves overhead that isn’t necessary when all processors are located in the same physical computer, so a new communications library that eliminates unnecessary overhead is necessary to achieve the greatest possible benefit from additional co-located processors. IPC is a communications library that provides shared memory segments that any process can attach to, making it an ideal mechanism with which to implement an Ironman library. Using IPC to successfully allow multiple processors to communicate is challenging, however, as the semantics for shared memory already implemented the SHMEM library are different enough from IPC that replacing shared memory function calls with IPC function calls is not straightforward. While shared memory via IPC is a promising implementation strategy, the use of threads may be more efficient as the OS kernel is not involved and memory does not need to be shared explicitly.

Acknowledgements
Thanks to Bradford Chamberlain for his invaluable assistance in deciphering the current Ironman implementations and his assistance in determining an approach to this problem, and to Larry Snyder for advising me through this long process. Thanks also to Nathan Kuchta for allowing me to gain some insight from his Ironman implementation.
References


program red_blue;

---------- Declarations ----------

config var
  n       : integer = 50;       -- board size
  x       : integer = 10000;
  timing  : boolean = false;    -- determine the execution time

region
  R    = [1..n, 1..n];          -- problem region

constant
  empty   : integer = 0;
  red     : integer = 1;
  blue    : integer = 2;

var
  count   : integer = 0;
  WORLD   : [R] integer;
  TEMP    : [R] integer;
  REDMASK : [R] boolean;
  BLUEMUASK : [R] boolean;
  MASK    : [R] boolean;

direction
  north  = [-1, 0];            -- cardinal directions
  east   = [0, 1];
  south  = [1, 0];
  west   = [0, -1];
  sw     = [1, -1];

---------- Init Procedure ----------

procedure init(var X: [ , ] integer); -- array initialization routine
  [R] begin
    if timing then
      ResetTimer();
    end;
    X := empty;
    [west in '] X := red;
    [north in '] X := blue;
  end;

---------- Entry Procedure ----------
procedure red_blue();
var
    runtime : double;
begin
    init(WORLD);
    repeat
        TEMP := WORLD;
        WORLD := empty;
        BLUEMASK := (TEMP = blue & TEMP@^south = empty & TEMP@^sw != red);
        REDMASK := (TEMP = red & TEMP@^east = empty);
        MASK := !(TEMP = blue & TEMP@^south = empty & TEMP@^sw != red) &
            !(TEMP = red & TEMP@^east = empty) &
            !(TEMP = empty & (TEMP@^west = red | TEMP@^north = blue));
        [R with MASK] WORLD := TEMP;
        [R with REDMASK] WORLD@^east := red;
        [R with BLUEMASK] WORLD@^south := blue;
        count := count + 1;
    until (x <= count);
    if timing then
        runtime := CheckTimer();
    end;
    if timing then
        writeln("Execution time: ", "%.6f":runtime);
    end;
end;