Parallelizing VCG2D/3D

Marco Adolfo Maurier

Problem Report submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in a partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Approved By

Francis VanScoy, Ph.D., Chair
James Mooney, Ph.D.,
Martin Ferer, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2011

Keywords: Open MPI, OpenMP, NVIDIA CUDA, multi-core

Abstract

Parallelizing VCG2D/3D

Marco Adolfo Maurier

The present availability of clusters, multi-core computers, and GPGPU's, brings new challenges to application developers and scientists seeking the best possible way to use such resources. Each one of these technologies has advantages and disadvantages depending on the nature of the program that the developer is trying to parallelize. The burden to achieve the highest possible performance is mostly on the compiler and/or the applications developer. Understanding of both the software and the hardware architectures of every tool involved in a project is essential for making correct decisions before starting any software design and coding.

This document will study several different scenarios for attempting to improve the performance of a FORTRAN program which simulates flow through a porous material. It was originally written as a sequential code in FORTRAN 77 and later on parallelization techniques were implemented using Open MPI. This code named VCG2D/3D was handed down to us for maintenance by Dr. Ferer. After careful analysis of the main loop, it can be concluded that the parallelization has design problems which lead to erroneous results. Also the degree of complexity of the VCG2D/3D brings to light the limitations of code parallelization techniques using clusters, where simple partitioning the arrays within the main loop, call Solns5, and sending them to different nodes actually decreases the program performance compared to the sequential code.

Three different approaches for parallelizing VCG2D/3D were analyzed, along with their advantages and disadvantages. The first one uses a Beowulf cluster using Open MPI, and the second a multicore processor also using Open MPI, and finally with a GPGPU (General Purpose Graphics Processing Unit) using NVIDIA CUDA. The platform that produced the best results was NVIDIA CUDA with a speed-up of 7.7 over the sequential code.

Table of Contents

Table of Figures

Index of Tables

Acknowledgements

- *To Dr. Martin Ferer* − Thank you for allowing me to work on your code
- *To Dr. Francis Van Scoy* − Thank you for use of the Virtual Lab access
- *To Dr. James Mooney* − Thank you for the valuable computer architecture lecture

FORTRAN ProgramVCG2D/3D

Introduction

Over the summer of 2008, I was asked by Dr. Ferer, a professor at physics department, to maintain two FORTRAN programs. These programs named VCG2D and VCG3D simulate fluid moving through a porous material. VCG2D and VCG3D had been parallelized by an undergraduate physics student using Open MPI. The approach taken divides the arrays in the subroutine Solns5 into several parts, sends each part to different processors, each processor would works independently, and finally the arrays are reassembled in the root node. Using this approach, nodes do not communicate the changes made in their respective arrays partitions.

Subroutine Solns5 in VCG2D

The bulk of the nested loops for both programs are in the subroutine Solns5. For the purpose of this study the loops were named as follow, loop30, loop3030, loop31, and loo3131. Most of the VCG2D/3D processing time is spent within Solns5; this is why all of the parallelizing with Open MPI was done to there. Loop30 and loop31 were written as nested loops so that may access selected cell within the arrays. Solns5 in VCG2D uses three two dimensional arrays named P, G, and GSUM. The loops G and GSUM remain unchanged in Solns5, on the other hand P stores the changes produced by two identical linear equations. Here is a code sample of Solns5. The green and yellow show the locations access in array P:

Table 1 Solns5 2D sample code

```
      DO 30 J=4,LASTW2,2
      J1=J-1
      DO 30 I=4+MOD(J,4),LASTL2,4
      I1=I-1
      PI=P(I,J)
      P(I,J)=(G(I1,J)*P(I-2,J)+G(I+1,J)*P(I+2,J)+G(I-2,J1)
     1*P(I-2,J-2)+G(I,J+1)*P(I+2,J+2)+GSUM(I1,J1))*(GSUM(I,J))
      PE=(PI-P(I,J))*(PI-P(I,J))
   30 ERROR=ERROR+PE
      DO 3030 I=4,LASTL2,2
      P(I,2)=P(I,LASTW2)
 3030 P(I,LASTW)=P(I,4)
      DO 31 J=4,LASTW2,2
      J1=J-1
      DO 31 I=4+MOD(MOD(J,4)+2,4),LASTL2,4
      I1=I-1
      PI=P(I,J)
      P(I,J)=(G(I1,J)*P(I-2,J)+G(I+1,J)*P(I+2,J)+G(I-2,J1)
     1*P(I-2,J-2)+G(I,J+1)*P(I+2,J+2)+GSUM(I1,J1))*(GSUM(I,J))
      PE=(PI-P(I,J))*(PI-P(I,J))
   31 ERROR=ERROR+PE
      DO 3131 I=4,LASTL2,2
      P(I,2)=P(I,LASTW2)
 3131 P(I,LASTW)=P(I,4)
      IF(ERROR.LT.DD)GO TO 99
```

During each iteration array P is accessed in four different cells, as shown by Figure 1. The *X* indicates *P(I, J)*.

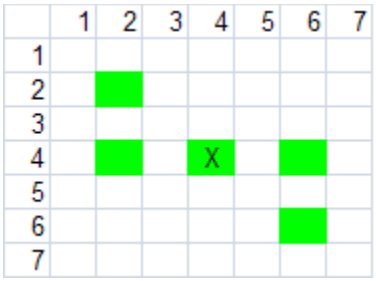

Figure 1. Single cell access for P

During the same iteration the changes made by the previous cell do not affect the next cell. This applies for loop30 and loop31. The data dependencies between loop30 and loop31 will be shown in Figure 5.

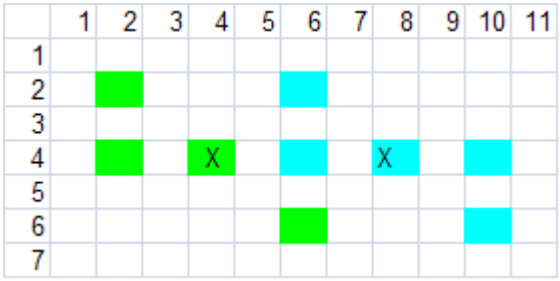

Figure 2. Two cells access for P

Starting with the second iteration in both loops, P will need the result from the previous iterations for cell *P(I-2, J-2)*. As shown in Figure 3, cell *P (4, 4)* takes the sum of the products and this result is used by *P (6, 6)* in the next iteration. Therefore all iterations use results from previous iteration with the exception of the first iteration.

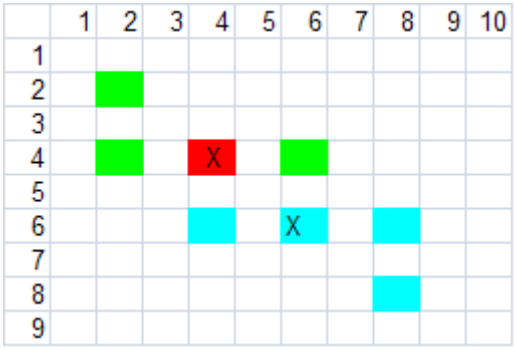

Figure 3. Cell access during next iteration

After loop30 is finished loop3030 copies values from the second to the last row and from the penultimate to the first row. The two rows being copied have new values in every other cell. In

loop31 most cells being changed will use two values that were modified by loop30. At this point loop31 has data dependencies between iterations and also from cells modified by loop30 and loop3031.

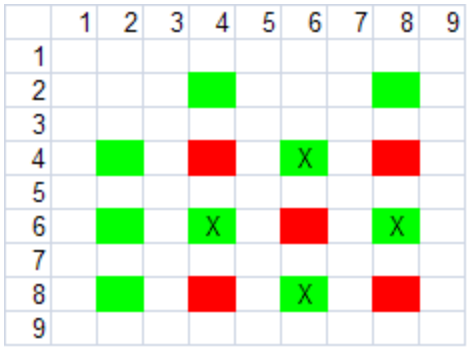

Figure 4. Data dependency between loop30/31

Subroutine Solns5 in VCG3D

Table 2 Solns5 3D sample code

```
      DO 30 i=2,LSTLH1
      DO 30 j=2,LSTLH1
      DO 30 k=2,LSTLH1
      PI=P5(I,J,K)
      P5(I,J,K)=(Gmi(I,J,K)*P5(I-1,J,K)+Gmi(I+1,J,K)*P5(I+1,J,K)+G(I,J,K)
     1*P5(I,J-1,K)+Gmj(I,J+1,K)*P5(I,J+1,K)+Gmk(I,J,K+1)*P5(I,J+1,K)+
     2Gmk(I,J,K+1))*(GSUM(I,J)*P5(I,J,K+1)+Gmk(I,J,K)*P5(I,J,K-1)+
     3+GSUMo(I,J,K))*(GSUMe(I,J,K))
      PE=(PI-P5(I,J,K))*(PI-P5(I,J,K))
   30 ERROR=ERROR+PE
      DO 3030 I=2,LstLH1
      DO 3030 n=2,LstLH1
      P5(I,1,n)=P5(I,LstWH1,n)
      P5(I,1,LstWH1,n)=P5(I,2,n)
      P5(I,n,1)=P5(I,n,LstWH1)
 3030 P(I,n,LASTW)=P(I,n,2)
     IF(ERROR.LT.DD)GO TO 99
```

In VCG3D loop31 and loop3131 are removed. All the arrays are three dimensional and data dependencies have increased. Every cell accesses its immediate neighbor in all dimensions. As a result every cell uses the results of the previous cell and the every iteration uses all results from the previous iteration as shown in Figure 6.

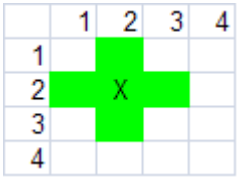

Figure 5. Single cell access for VCG3D

Starting with the second iterations most cells will need the results from the previous cell and the previous iteration. Figure 7 shows the data dependency for cell P5(3,3,3) where it needs the values from P5(2,3,3), P5(3,2,3) and P5(3,3,2) (not shown).

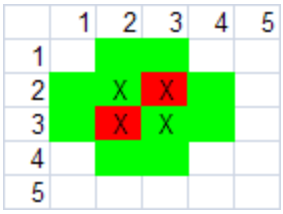

Figure 6. Data dependencies in second iteration in one dimension

Previous parallelization of VCG2D/3D

The original approach for parallelizing VCG2D/3D was to divide the array in blocks proportional to the amount of processors, create ghost columns next to the bordering cells, make the calculations independently in processors, and recalculate neighboring cells after joining the array after all processors finish working in the block of the array they were assigned. Any data dependencies between bordering cells were ignored while processing the individual blocks.

Ideally the results from every P cell that is changed should be the same or very close in both the sequential and parallel programs otherwise the parallelization is incorrect. Figure 8 shows the data dependencies between neighboring cells in VCG2D. In this figure we consider the interaction between two processors. P1 and P2 represent processor one and processor two respectively. During the second iteration P2(16,6) will need the results of P1(14,4). The results of P2(16,6) will be used during the next iteration by P2(18,8), and the next iteration P2(18,8) will be use by P2(20,10) and so on. By ignoring the data dependencies between neighboring cells we will produces results that are not identical to the results reached by the sequential code version of VCG2. In order to produce an accurate parallelized flow model, all processing units must communicate cell changes that will be used by elements in other processing units. This communication must be synchronized during each of the iterations.

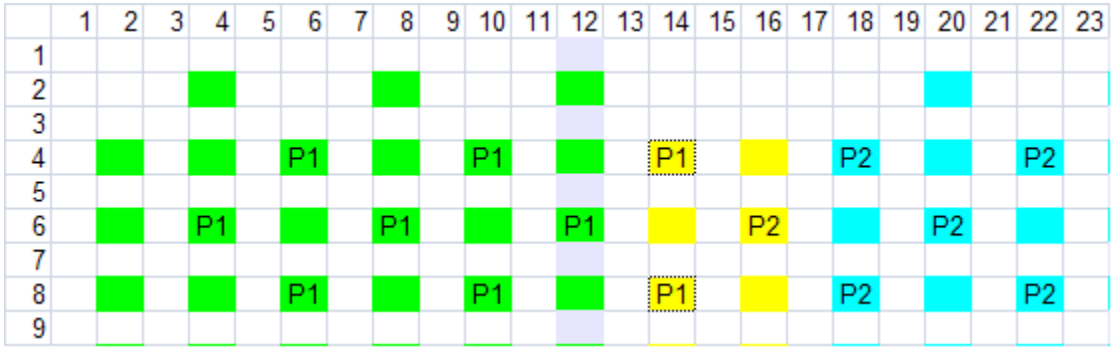

Figure 7. Data dependency between two processors in loop30/31 shown in yellow

Parallelizing Using Open MPI

Clusters and VCG2D/3D

The challenge that Open MPI faces with VCG2D/3D is in the fact that that all portions of the array distributed among the nodes must communicate their results during or after each iteration. The communication of message is a difficult endeavor for a developer due to the synchronization required between all messages. The delays caused by the combination of communication and synchronization will slow down the program performance to the point where the sequential code will actually run faster. Also worth mentioning is the fact that debugging the synchronization among nodes is a long and difficult process and achieving correct results is very time consuming.

Home Cluster

When the FORTRAN code was assigned to me I had no access to a cluster. So in order to study VCG2D/3D under Open MPI I setup a four node cluster at home using old Pentium computers, and a Linksys switch. The cluster software used was OSCAR (Open Source Cluster Application Resource) running under Linux Fedora 5. The first node acted as head-node and the other three as root node, node 1, and 2. VCG2D produces a number of text files with different outputs, for the purpose of this study all outputs to files were removed except one. The outputs write to the screen the number of iterations reached and the error found, before exiting Solns5. This numbers are important because they determine whether the MPI code is produce identical result as the sequential code.

The progress of the sequential code into parallel MPICH code was done by creating subdirectories for each new version. By the time the correct version was produced there were 76 version of VCG2D, the last version being 32.6.4. This last version works exactly the same as the sequential.

Open MPI Code

The communication among processes was done with mpi_send() and mpi_recv() commands. In order to collect the errors we used mpi_allreduce(). The mpi_bcast() command was used to broadcast the arrays values at the start and at the end of the loop. Work among processors was divided by using their individual ranks. Table 3 shows a sample code for rank one and loop30. There is a send/recv message for every iteration, for every block, and through the whole loop. Blocks with neighbors on both ends of the loop will have a sen/revc message for each end. Basically all processors except the first and last have two send/recv call per iteration. Another operation that requires communication from all processes is the mpi_allreduce(). This call waits for all ranks to transmit their individual errors, once it has received all messages, and then it will sum them.

Table 3 MPI sample code for rank 1 in loop30

```
      IF(irank.eq.1) then
      DO 30 j=4,lastw2,2
      j1=j-1
      DO 30 i=4+mod(j,4),34,4
      IF(mwtot.gt.1.and.i.eq.34)p(i+2,j) = colAdj1(j)
      i1=i-1
      PI=P(I,J)
      P(I,J)=(G(I1,J)*P(I-2,J)+G(I+1,J)*P(I+2,J)+G(I-2,J1)
     1*P(I-2,J-2)+G(I,J+1)*P(I+2,J+2)+GSUM(I1,J1))*(GSUM(I,J))
      PE=(PI-P(I,J))*(PI-P(I,J))
      errorloc=errorloc+pe
      if(i.eq.34) then
      sndRnk1 = p(i,j)
      CALL MPI_SEND(sndRnk1,1,mpi_double_precision,2,tag,
     1mpi_comm_world,ierr)
      end if
   30 continue
      CALL MPI_RECV(colAdj1,lastw2,mpi_double_precision,2,tag,
     1mpi_comm_world,stat,ierr)
```

Results

The results from the MPICH modified VCG2D code was disappointing. The sequential code runs in just 7.7 seconds while the parallelized version runs in 77 minutes. The MPI code was also run in a cluster at the Virtual Lab producing similar results. There is a great amount spent trying to synchronize messages and debugging errors produce at run time are also very time consuming due to the fact that we are dealing with several machine that are accessed indirectly by means of the root node.

The level of node intercommunication needed in order to keep the results accurate is such, that performance is sacrificed. Every node has to send and receive data during each iteration, and before and after each loop. The value of `ERROR` has to be reduced after the loops are done, which means that all nodes send their `ERROR` values to the root node where is reduced.

This outcome demonstrates limitation on the degree of parallelization that can be accomplished using Open MPI with the average hardware available to users in a case where there is heavy intercommunication among nodes. Another point worth noting is the fact that VCG2D/3D use relatively small arrays, which makes it more difficult to beat the instruction level parallelism found in compilers and CPU architectures.

Parallelizing Using Multi-core

Multi-core and VCG2D/3D

The high-performance computing (HPC) community has been programming parallel computers using MPI for over 20 years. How will this technology do with multicore? These supercomputing clusters differ from the multiprocessor in a fundamental way: the memory in the system is distributed, not shared. This means that each processor has its own block of memory, and if we want to communicate something from one processor to another, we must use a buffer and call the MPI functions to send the message.

The VCG2D/3D code does not run well on distributed memory. Shared memory may help because of the fact that all communication among processors will take place using local RAM instead of network interconnection hardware. There is a downside to Open MPI running on multicore systems and is the fact that it suffers from many of the same problems as Pthreads and WinAPI threads. Actually, Open MPI has the same function in distributed-memory that Pthreads and WinAPI threads have in shared-memory multicore systems. What we may know wonder is if since MPI will be utilizing Pthreads and WinAPI threads, then does this mean that Open MPI will add overhead to the job distribution among processes or will complement it and in fact improve the performance of the code. As we will see this depends entirely on the relationship among the system hardware, the OS, and Open MPI.

The Mac Pro Architecture

Dr. Ferer has a Mac Pro workstation where he runs his code sequentially. We wanted to find out whether Open MPI would help us run VCG2D/3D faster. We need to understand the Mac Pro internally in order to see what it is capable of doing on its own. The Mac Pro has two Quad-Core Intel Xeon "Nehalem" processors. Each processor is a single-die, 64-bit architecture makes 8MB of fully shared L3 cache readily available to each of the four processor cores. The result is fast access to cache data and greater application performance.

The Mac OS X is based on the Mach kernel which was developed at Carnegie Mellon University. It is microkernel developed to research distributed and parallel computation. One critical difference between Mach and UNIX is the use of the pipe abstraction to replace the everything-is-a-file concept which is not as flexible. Pipes use secured ports for inter-process, task, and thread communications. Mach relies on the machine's memory management unit (MMU) to rapidly map the data from one process to the other. Essentially Mach was design to support multiprocessing systems and shared memory.

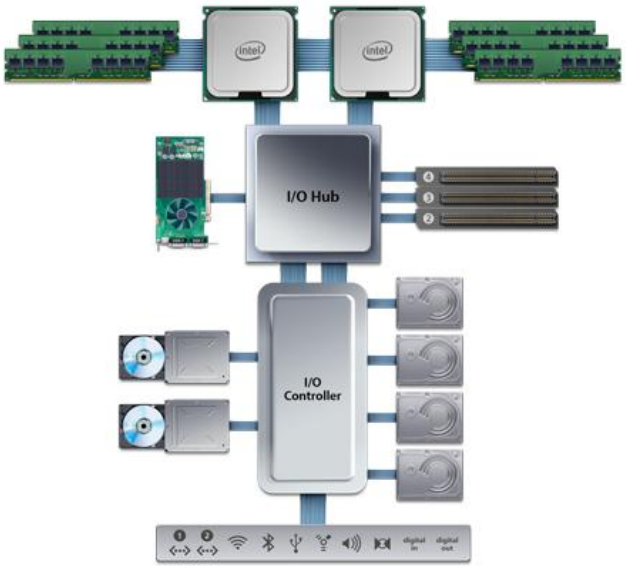

Figure 8. Mac Pro hardware architecture

Another remarkable Mac Pro feature is the fact that each Intel Xeon "Nehalem" processor features an integrated memory controller (Figure 9). So the RAM is directly connected to the processor. This means faster access to data stored in memory, and memory latency reduced by up to 40 percent. The integrated memory controller increases the memory bandwidth which allows for more data to be fed to the processor faster, helping each core spend its time processing data, not waiting for data to arrive. Finally in order to maintain cache coherence between the two processors the Intel 500P Memory Controller Hub (MCH) uses a Snooping algorithm to transfer data between cores.

Open MPI Code

We used the same VCG2D code modified with Open MPI to run on a three node cluster and changed it so that the main SOLNS5 loop is divided in 8 parts, one part per MAC Pro processor core. This was done by using the first and last core to process the first and last part of the array, and the cores in between to process the rest of the array. Open MPI should be able to create threads that are distributed to the eight cores hopefully make the code run faster than the sequential program.

Results

The sequential program using a 36x36x36 array finished in 17 sec. The Open MPI code ran in 1 minute 22 seconds. This makes the sequential code 11.6 times faster. The reason for the poor performance of the Open MPI code is due to the MAC OSX hardware and software architecture design. Open MPI creates an overhead head that makes the program run slower. The sequential program is processed by the OS distributing the instructions efficiently to the eight cores.

Parallelizing Using GPGPU and NVIDIA CUDA

Introduction

CUDA (Compute Unified Device Architecture) is a parallel computer architecture developed by NVIDIA. It uses an extended ANSI C programming language that allows developers to use a CUDA enabled GPU as a general-purpose processor for data parallel computation. At its core are three key abstractions thread groups, shared memories, and barrier synchronization that are handled by the programmer with a simple set of language extensions. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. This allows the programmer to partition the problem and enable transparent scalability since each problem can be scheduled to be solved on any number of the available processor cores and only the runtime system needs to know the physical processor count.

GPU Architecture

There is a significant difference between CPU and GPU architecture as shown by Figure 10. The GPU needs less sophisticated flow control for its operations because the same program is executed on many data elements. This also lessens the requirements for a large cache. The design follows the SIMD (single instruction multiple data) model, but NIVIDIA describe it a SIMT (single instruction multiple thread) model.

Figure 9. CPU vs. GPU architectures

Under CUDA each data element that can be parallelized is assigned to a thread. Each thread is grouped in Blocks, and finally blocks are grouped in a Grid as shown in Figure 11. This level of abstraction is what the programmer deals with. The number of Blocks, Threads and the size of the Grid are programmable. The GPU will map each of thread to one scalar processor core, and each thread executes with its own instruction address and register state. These threads are schedule in groups of 32 called warps.

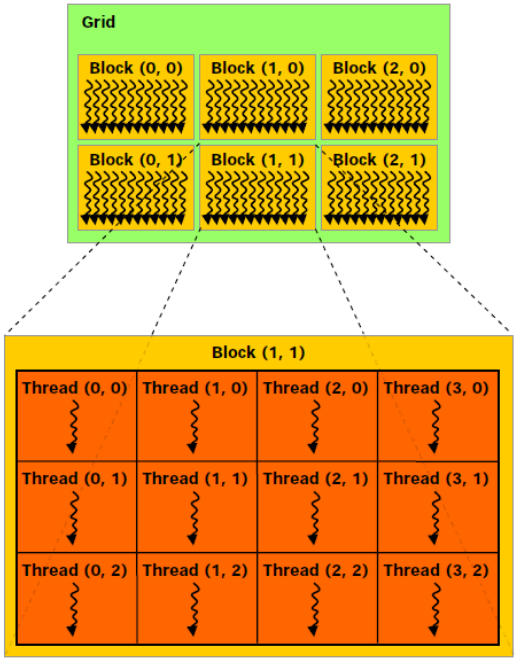

Figure 10. Threads, Blocks, and Grids

Home GPU

We saw the possibility of running VCG2D/3D as something that could be accomplished in the near future once NVIDIA CUDA comes up with a FORTRAN compiler for its GPU's. In order to test the VCG2D/3D code in a GPU first we had to construct a machine with Quad Core AMD, 1GB or RAM, and a CUDA enable device. The system was built using a mid-tower with an NVIDIA Geforce 9800 GX2 graphics card, which has two GPU's each with 128 streaming processors, a total of 256 cores, and one gigabyte of RAM. Due to time constraints only the subroutine SOLNS5 was used for testing.

Implementing NVIDIA CUDA

In order to run SOLNS5 in a CUDA enabled device, it had to first be translated to C. The main loop as whole was executed in the device. Before doing this all variables, including arrays must have memory allocated in the device memory and then they can be copied to the device. After this a function called the kernel is called. This function is executed in the host computer and calls the portion of code present in the device. A sample of the memory allocation and copy CUDA code is shown below.

Table 4 Cuda memory allocation and copy

```
cudaMalloc( (void **) &d_p, memSize );
cudaMalloc( (void **) &d_g, memSize );
cudaMalloc( (void **) &d_gsum, memSize );

cudaMemcpy( d_p, h_p, memSize, cudaMemcpyHostToDevice );
cudaMemcpy( d_g, h_g, memSize, cudaMemcpyHostToDevice );
cudaMemcpy( d_gsum, h_gsum, memSize, cudaMemcpyHostToDevice );
```

The code below show how the kernel is called. Once the kernel is finshed the result are copied back to host memory, and if nessesary all device memory is deallocated.

Table 5 Cuda kernel call

```
solns5Kernel<<< dimGrid, dimBlock >>>(d_p,d_g,d_gsum);

cudaMemcpy( h_p, d_p, memSize, cudaMemcpyDeviceToHost );

cudaFree(d_p);
cudaFree(d_g);
cudaFree(d_gsum);
```

Results

Due to the fact that VCG2D has data dependencies in between iterations, the only way to parallelize loop30/31 was to assign a single thread per array cell and execute one row at a time. This configuration does not take advantage of the possibility of using hundreds of threads simultaneously. Consequently the speed gain was not significant. The sequential VCG2D with an array of 96 by 260 took 64minutes and 16 seconds. The CUDA simulation of 96 by 262 took 2754 seconds or 46 minutes. This is speed-up of 40% over the sequential code. This figure will increase significantly with VCG3D once the code is redesign to take advantage of CUDA's thread parallelism.

Parallelizing Using OpenMP

Introduction

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared memory multiprocessor programming in C, C++, and FORTRAN. It is implemented by a series of compiler directives, library routines, and environmental variables. OpenMP uses multithreading where each thread executes the parallelized section of code independently. During runtime threads are allocated to processors depending on usage. We this in mind we used OpenMP to try to improve the performance of VCG3D.

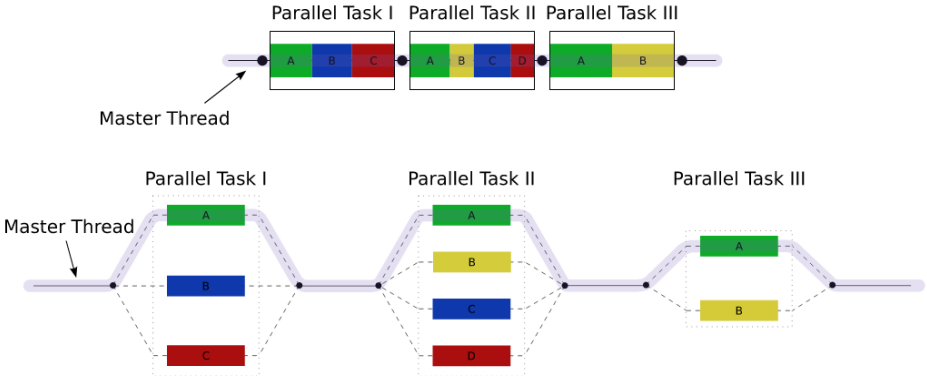

Figure11 Multithreading with OpenMP

Implementing OpenMP

In order to apply OpenMP to VCG3D we had to translate the code to FORTRAN 90. The translation work involved mostly changes the configuration of the code. All COMMON blocks were removed and variables were declared in a MODULE named VARS. In place of the COMMON block a USE VARS declaration is used. Also FORTRAN 90 does not have the margin restrictions that FORTRAN 77 has.

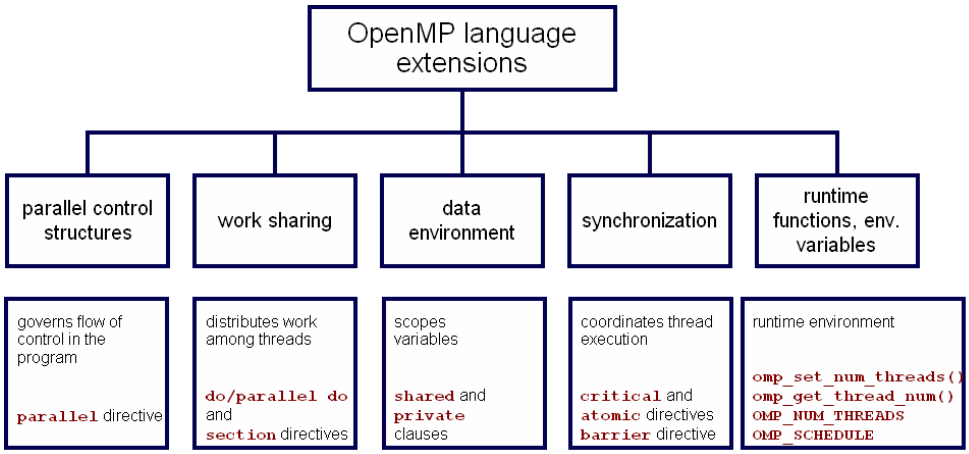

Figure 12. Chart of OpenMP construct

The implementation of OpenMP was very straight forward and did not take nearly as much time as with Open MPI. As Figure 14 shows parallelization is accomplished by using directives. Some variables are made private and by default the rest of the variables are shared. The directive REDUCTION(+:ERROR) tells OpenMP to sum all results from each process for the variable ERROR. All arrays are accessed by every process using the computer's shared memory. The first line is call that tells Open MP the number processes to assign for the parallelization.

The next line is directive that determines where the parallel code starts. Next we declare which variables will be private to each thread. The `!$omp do` tells the compiler that a do loop follows. This directive is ended by `!$omp end do` followed by `!$omp end parallel` which ends the OpenMP block.

Table 6 Sample code from SOLNS5 using OpenMP

```
      call OMP_set_NUM_THREADS(8)
      !$omp parallel &
      !$omp private(i,j,k,PE,PI)  reduction(+ : ERROR)
      !$omp do
      DO 30 i=2,LSTLH1
      DO 30 j=2,LSTWH1
      DO 30 k=2,LSTWH1
      PI=P5(I,J,K)
      P5(I,J,K)=(Gmi(I,J,K)*P5(I-1,J,K)+Gmi(I+1,J,K)*P5(I+1,J,K) &
     +Gmj(I,J,K)*P5(I,J-1,K)+Gmj(I,J+1,K)*P5(I,J+1,K)+Gmk(I,J,K+1) &
     *P5(I,J,K+1)+Gmk(I,J,K)*P5(I,J,K-1)+GSUMo(I,J,K))*(GSUMe(I,J,K))
      PE=(PI-P5(I,J,K))*(PI-P5(I,J,K))
   30 ERROR=ERROR+PE
      !$omp end do
      !$omp end parallel
```

OpenMP on the Mac Pro

Since OpenMp is an implementation of multithreading, a method of parallelization where the mater thread, which are defined as a series of instructions executed consecutively, forks a specified number of slave threads and a task is divided among them, the challenge of OpenMP will be to improve over the already optimized MAC OSX. These threads must then run concurrently, with the runtime environment allocating threads to different processors while they are distrusted by the scheduler.

Results

The sequential code with an array of 36x36 finished running in 3 minutes and 7 seconds. The OpenMP code finished running in 2 minutes and 39 seconds. This is a 60% speed up. This results show some improvement of Open MPI. The overhead of OpenMP for synchronization and loop scheduling are the important factor in determining the performance of shared memory while running VCG3D. Even though Open MPI is a more advanced protocol it is better suited for cluster configurations than for multicore processors.

Parallelizing VCG3D Using ILP

Introduction

Before we can determine how much parallelism exists in a program and how it can be exploited we must look at the instruction-level parallelism (ILP), or how the instructions can be executed simultaneously in a pipeline without causing any stalls. In the case of VCG3D we are trying increase the loop-level-parallelism among iterations in loop31.

A data hazard is when data dependencies are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. The sample code below shows three apparent data hazards among four lines of code. First we have WAR (write after read), or anti-dependency, between lines 1 and 2. Second is RAW (read after write), or true data dependency, between line 2 and 3, and between line 4 and 3.

Table 7. Data dependence in SOLNS5

```
1     PI=P5(I,J,K)
2     P5(I,J,K)=(Gmi(I,J,K)*P5(I-1,J,K)+Gmi(I+1,J,K)*P5(I+1,J,K)+…
3     PE=(PI-P5(I,J,K))*(PI-P5(I,J,K))
4  30 ERROR=ERROR+PE
```

After analyzing the loop-level parallelism of SOLNS5 we can determine that in addition to this data hazards there are also loop-carried dependence within P5. This means that data accesses in later iterations are dependent on data values produced in earlier iterations as shown by the areas in line 2 with yellow highlights. The anti-dependency between line 1 and 2 is most likely handled by the compiler by performing `GSUMo(I,J,K))*(GSUMe(I,J,K)` after PI is loaded with P5 in line 1. This would eliminate the first data dependence. Lines 2, 3, and 4 overlap during execution and the order cannot be changed, this makes their ILP limited.

ILP on the Mac Pro

Due to the way that P5 was being processed in SOLNS5 we tried a new approach to accessing every cell in P5 in order to reduce the data dependence. Loop30 was split eight sequential loops. The collected ERROR was close enough to the original unmodified loop. In order to reduce the data dependency in SOLNS5 loop30 was transformed.

Table 8 Sample code for eight part loop

```
!************************* 8 part loop start ***************

      DO 301 i=2,LSTLH1,2
      DO 301 j=2,LSTWH1,2
      DO 301 k=2,LSTWH1,2
      PI=P5(I,J,K)
      p5temp(1) = GSUMo(I,J,K)*GSUMe(I,J,K)
      p5temp(2) = Gmi(I+1,J,K)*P5(I+1,J,K)
      p5temp(3) = Gmj(I,J,K)*P5(I,J-1,K)
      p5temp(4) = Gmj(I,J+1,K)*P5(I,J+1,K)
      p5temp(5) = Gmk(I,J,K+1)*P5(I,J,K+1)
```

```
      p5temp(6) = Gmk(I,J,K)*P5(I,J,K-1)
      p5temp(7) = Gmi(I,J,K)*P5(I-1,J,K)

      P5(I,J,K) = sum(p5temp)
      PE=(PI-P5(I,J,K))**2


      er(1)=er(1)+PE
  301 continue

      DO 302 i=3,LSTLH1,2
      DO 302 j=2,LSTWH1,2
      DO 302 k=2,LSTWH1,2
      PI=P5(I,J,K)
      p5temp(1) = GSUMo(I,J,K)*GSUMe(I,J,K)
      p5temp(2) = Gmi(I+1,J,K)*P5(I+1,J,K)
      p5temp(3) = Gmj(I,J,K)*P5(I,J-1,K)
      p5temp(4) = Gmj(I,J+1,K)*P5(I,J+1,K)
      p5temp(5) = Gmk(I,J,K+1)*P5(I,J,K+1)
      p5temp(6) = Gmk(I,J,K)*P5(I,J,K-1)
      p5temp(7) = Gmi(I,J,K)*P5(I-1,J,K)

      P5(I,J,K) = sum(p5temp)
      PE=(PI-P5(I,J,K))**2

      er(2)=er(2)+PE
  302 continue

.
.
.
 ERROR = sum(er)
```

Results

The VCG2D code with its SOLNS5 subroutine main loop divided in 8 parts finish running in two minutes and nine seconds. The sequential version finish running in three minutes and seven seconds. The modified eight part loop was 1.6 times faster than sequential. These results show that the modifications done on SOLNS5 without using OpenMP will produce the same result that OpenMP will produce. The fact that the MAC OSX kernel will improve its pipe line processing is evident with these results.

Parallelizing VCG3D Using NVIDIA CUDA

Introduction

After modifying VCG3D by dividing loop30 in SOLNS5 we came to the conclusion that the output from the original VCG3D and the modified one where close enough to be acceptable. As a consequence of these results we decided to implement SOLNS5 ignoring the data dependencies among cells until after entire 3D array has been processed. This way cells have access to new cell values only after a complete 3D iteration is finished. As a consequence of this configuration each cell in the 3D array can be assigned to a single thread.

Implementing CUDA

The mechanisms to allocate, copy memory, and start the kernel are the same as the ones previously mentioned. Even though the device claims to have 1 GB of RAM, only 512 MB are available to the user. This brings limitations to size of the array that can be loaded into the device. Another aspect worth mentioning is the fact that the Geforce 9800 GX2 graphics card is a device with compute capability 1.1, which means that the GPU cannot process double-precision instructions.

The SOLNS5 subroutine exits when the variable ERROR reaches a certain value. The number of times loop30 is accessed before this happens is on the average three to five thousand times. In order to come to an approximation of what VCG3D does when it calls SOLNS5, we called SOLNS5 once, and ran loop30 five thousand times; once for the CUDA device and once for the sequential host code. The running time was measured using a CUDA timer tool and the speed-up was obtained by dividing the sequential time by CUDA device time.

All arrays were loaded in the device local memory, but to improve the performance of the arithmetic operations all variables were transferred to the device shared memory. The only moment when synchronization is an issue is when values a copied from local to shared memory or vice versa. This is accomplished with kernel command `__syncthreads()`. Shared memory declared within the kernel and it is almost as fast a register memory. Below are sample code lines of the kernel declarations.

Table 9 Cuda 3D Sample Code

```
__shared__ float var [14], result, pe, pi;

if(validRange)
      for(j=1;j<SIZE-1;j++)
            var[6]=g_gmi[i+1+j*SIZE+k*SIZE2]; //Gmi(I+1,J,K)
            var[7]=g_gmj[i+(j+1)*SIZE+k*SIZE2]; //Gmj(I,J+1,K)
            var[8]=g_gmk[i+j*SIZE+(k+1)*SIZE2]; //Gmk(I,J,K+1)
            var[9]=g_gmi[i+j*SIZE+k*SIZE2]; //Gmi(I,J,K)
            var[10]=g_gmj[i+j*SIZE+k*SIZE2]; //Gmj(I,J,K)
            var[11]=g_gmk[i+j*SIZE+k*SIZE2]; //Gmk(I,J,K)
            var[12]=g_gsumo[i+j*SIZE+k*SIZE2]; //GSUMo(I,J,K)
            var[13]=g_gsume[i+j*SIZE+k*SIZE2]; //GSUMe(I,J,K)
            __syncthreads();
```

The equation in SOLNS5 caused read memory conflicts among threads. The cells involved in the equation in one thread are also used by other threads. This fact will cause memory reads to fail due to synchronization conflicts. So we removed the `__shared__` option and loaded all variable on the global memory in order to remove thread coalitions.

The variable must collect all errors calculated in every cell of the 3D array in order determine whether it needs to exit SOLNS5 or not. When using CUDA the error can be calculate per thread, and then each block of thread has to have a single to calculate or reduce all errors from every thread within the block. Finally in order to reduce the error calculated by all blocks, a block and a thread within that block is selected within the grid to reduce all errors found in every single block. As shown in the sample code below.

Table 10 CUDA code for error reduction

```
__device__ float error_block [gridDim.x][ gridDim.x];
__shared__ float error_thread [blockDim.x][ blockDim.x];

error_thread[threadIdx.x][threadIdx.y] = error_t;

if(threadIdx.x == 0 && threadIdx.y == 0)
{
   for(int x=0;x< blockDim.x;x++)
          for(int y=0;y< blockDim.x;y++)
                 error_block[blockIdx.x][blockIdx.y] += error_thread[x][y];
   __syncthreads();
}

if(blockIdx.x == 0 && blockIdx.y == 0 && threadIdx.x == 0 && threadIdx.y == 0)
{
   for(int x=0;x< gridDim.x;x++)
                 for(int y=0;y< gridDim.x;y++)
                        error_final += error_block[x][y];
 }
```

The array `error_thread[][]` had to be made type `__shared__` in order for it to be accessible only to all threads in the block. The array `error_block[][]` had to be made type `__device__` in order to be accessible to all blocks in the grid.

Results

The CUDA program was compared to the sequential time by measuring the running time for a single iteration to the entire 3D array. The time to allocate device memory and copy to the device all of the 3D arrays used by SOLNS5 is 41 milliseconds. A single iteration of the sequential code takes 21 milliseconds. The CUDA device takes 43 ms. including iterations through the 3D array and including error reduction with memory coalescence. It takes 50 ms. without memory coalescence.

As mentioned before on the average SOLNS5 is access 5,000 before exiting. So will added a 5000 iterations loop to both, the CUDA program and the sequential program. We'll call this loop the MWSNS loop. CUDA memory allocation and copy are what consumes the most time. As we include the MWSNS loop in SOLNS5, the CUDA kernel code outperforms the sequential code. The CUDA device takes 15396.28 ms. with the MWSNS loop. The sequential code takes 118510.79 ms. with the MWSNS loop. The speedup of CUDA over sequential is 7.7.

Final Thoughts

VCG2D/3D has inherited data dependencies in arrays accessed by Solns5 that show some limitations of today's clusters, and parallel libraries. In this paper we studied Beowulf clusters, multicore processors, and GPU's using Open MPI, Open MP, ILP, and CUDA. Clusters have data transmission delays that produce a performance which is below the performance of Mac Pro running OSX. Shared memory protocol like Open MP also is not able to improve the performance of the built-in distributed algorithms is the Mac Pro. Instruction Level Parallelism techniques show some gains in the speed of the program but nothing to significant. CUDA was able to speed up the running time of Solns5 by 7.7 but only after we modified the loop so that it ignores data dependencies until the following loop.

References

(1) Crossover from Capillary Fingering to Compact Invasion for Two-Phase Drainage with Stable Viscosity Ratios, Martin Ferer, Grants S. Bromhal, and Duane H. Smith, Advances in Water Resources in press (2006)
(2) Parallel Programming with MPI, Peter Pacheco
(3) Using MPI: Portable Parallel Programming with MPI, William Gropp
(4) Parallel Programming win C with MPI and OpenMP, Michael J. Quinn
(5) Parallel and Distributed Programming Using C++, Cameron and Tracey Hughes
(6) Computer Architecture: a Quantitative Approach, John L. Hennessy, David A. Patterson
(7) High Performance Linux Clusters with OSCAR, Rocks, and MPI, Joseph D. Sloan
(8) Beowulf Cluster Computing with Linux, William Gropp
(9) NVIDIA CUDA Programming Guide 2.3