Optimization of a Dynamic Random Surface Code for RISC Processors

Leping Han

Syracuse Center for Computational Science
and
Department of Physics
Syracuse University,
Syracuse, NY 13244, USA

1 Introduction

1.1 Issues about Computer Hardware and Software

With dramatic changes in technology ahead, how do we approach the problem of high-performance architecture design and high-performance engineering and scientific computing? For example, the new technology makes feasible massive parallelism. How much additional effort should be invested in increasing the performance on a single processor before we seek higher levels of performance on multiple processors? There are no simple answer to these questions. The latter is always essential to the first. We need a combination of solutions, and what we choose almost certainly will be application dependent, since at this stage we have not constructed a general machine that would be equally effective for all high performance applications yet.

In the past, we have seen many different techniques used in hardware to improve performance, such things as instruction buffers, cache memories, pipelined execution and RISC computer architecture [1, 2], have appeared in many commercial machine implementations. However, can a application well and easily utilize these hardware capabilities? This is always a complicated story. In terms of software issues, it has to address to both the level of compiler design and the level of user's applications and algorithms. As computer clock speed keeps on increasing, the issue of how well applications can utilize a computer's cycle becomes even more important, since many numerical applications are so demanding of computational cycles and call for sophisticated floating-point processors in the architecture [3]

1.2 Introduction to the Problem and the Program

Unstructured numerical applications [4, 5] have been one of the most computationally demanding areas and have the most complicated but the richest data structures. Naturally people like to use them to evaluate the overall performance of a computer system in dealing irregular problems for both hardware and software aspects [4, 6].

The Monte Carlo simulation of dynamically triangulated random surfaces happens to be one of the most suitable application to be investigated for state-of-the-art computers. It has very irregular with complicated data structures (*Figure* 1) and demands very intensive floating point calculations. The computation is very time consuming. For example, running on HP 9000 24 hours a day, the computation takes about 3 months to have one statistically meaningful physical data point for a mesh with 1152 nodes, even though our code is the fastest sequential code for dynamical random surface simulation.

Before we go on, we need to explain a little more about the physical problem and its meaning.

In physics, string theory, in a number of guises, has been conjectured to describe the underlying fundamental physics of a wide variety of physical phenomena and models. These include the strong interaction at long distances, the three-dimensional

Ising model and unified models incorporating gravity [7]. The theories are candidates for a TOE (Theory of Everything). In its simplest form, the bosonic string, it is a theory of free fluctuating surfaces (i.e. random surfaces). The functional integral for the Euclideanized bosonic string is just the partition function for an ensemble of random fluctuating fluid surfaces. Calculations involve integrating over all possible world-sheets, i.e. all 2-d surfaces embedded in some higher dimensional space (3-d, 4-d, 26-d, ...) [8, 9, 10] (see *Figure* 2 and 3 random surfaces, smooth and crumple)

Much progress in numerically simulating strings has been made through the use of Dynamically Triangulated Random Surfaces [8]. Interesting critical points are phase transitions between a *smooth* phase (*Figure* 2) and a *crumpled* phase (*Figure* 3) of the surface with extrinsic curvature [11, 12, 13]. By last December, we had done the largest simulations and most accurate measurements of physically interesting observables so far on 2-d systems, but only with 576 node mesh [13]. We are now running on even larger meshes with 1152 and 2304 nodes. This may show more conclusive physical results. However, critical slowing down is a major problem. The simulation takes hundreds of thousands of sweeps of updating the mesh to yield one statistically independent configuration. The existence order and critical exponents of this *crumpling transition* are still uncertain. That is our driving force to speed up the code by optimizing it, looking new algorithm and using parallel computer. The study of computational behavior of the random surface simulation in computer aspect is also very important, since it is a real and very typical irregular problem. It provides a basis for on-going High Performance Fortran compiler support [6].

2 The Program

2.1 The Fortran Code

The program used for the Monte Carlo simulation of string theories was written in Fortran. The following outlines the crucial part of main program.

```
program main-outline

CALL mesh-set-up
CALL global-data-initialization

DO i = 1, n_measurements
  DO j = 1, n_sweeps
    CALL update-nodes
    CALL update-links
  ENDDO
  CALL measurements
ENDDO
```

END

The program first constructs a initial mesh. *Figure* 4 shows a initial mesh with torus topology. We can see that one sweep consists of two parts : *update-nodes* and *update-links* . The subroutine *update-nodes* is used to update each node's position of mesh in the three dimensional space (*Figure* 5). The subroutine *update-links* is then used to change the connectivity of the mesh, i.e., to try to flip each link between nodes in the mesh (*Figure* 6). The Metropolis algorithm is used to update both the nodes and links of the surface. As the simulation proceeds, the geometrical data and the physical observable associated with nodes and edges have to be recalculated.

The node update part loops over all nodes. In the loop it randomly picks an attempted position for the current node and calculates the energy change in the simulated system between the old position and the attempted position (*Figure* 5). The Metropolis algorithm determines if the attempted position should really be used to update the node's position or not. The energy calculation is based on the length of the edges and the so called edge action, i.e. the dotproduct of two normal direction vectors of two triangles along the common edge. This dotproduct is actually calculated in terms of the two altitudes in the triangles along the common edge. The edge action calculation is very expensive since the calculation not only involves the data associated with all nearest neighbors but also the next nearest neighbors (*Figure* 7). However the node update is still less irregular in the sense that the connectivity is not changed. Therefore the data retrieving process from cache or main memory for the calculation of node update is still relatively smooth. It probably does not experience very frequent interruptions or jumps since hopefully most of the required data are laid out in memory consecutively, so useful data may stay in the cache for a while before moved out. With fixed connectivity, there are fewer branchings and fewer pointers to cause the memory jumps than in subroutine *update-links* where the connectivity is changing as each link flips. Therefore *update-nodes* produces fewer cache miss and page fault than *update-links* . However, even in the case of update node, the mesh is still irregular. Each node may have different number of neighbors, and therefore a different number of edges. These complexities may cause load balance problems for later parallel computation.

The subroutine *update-links* is used to randomly pick a link and attempt to flip it to another diagonal as in *Figure* 6. The calculation of the energy change between the attempted configuration and old configuration is still similar to the one in *update-nodes* which uses the Metropolis algorithm. However at this time, not only the calculation is as expensive as the *update-nodes* case, but also the data structure layout in the memory associated with the nodes and edges may become very irregular. To calculate observables, we may have to look for them throughout a big area of memory since the pointers may point to different places as we add links to one node and remove links from others. The data associated with the changing edges are scattered all over the memory and logically closely related data information are moving far apart in memory as the links flip. Data locations are less predictable (This will be explained more later in the paper). Therefore, as the simulation proceeds, the data associated with the changing edge may be moved in and out from the cache more frequently, and the data required to calculate observables may be distributed over different pages in virtual memory space. Therefore the behavior of *update-links* part may appear to be more "violent" in terms of memory access and becomes more interesting.

All of these characteristics are very good features to be used as a probe to investigate the performance of application software, compiler and memory hierarchy. This is because one code has two typical modes of behavior. By monitoring the performance of the two parts, we may gain some insights into many computational issues.

2.2 A comparison with a C Code

At the initial stage of studying random surfaces, we used a program which was written in C by Baillie, Johnston and Williams [14, ?], and was based on DIME (Distributed Irregular Mesh Environment), a general software package for handling dynamically triangulated meshes [5]. Dynamic unstructured meshes of this type are also used for finite element simulations, computational fluid dynamics and many other areas. The generality of the code meant that it used a lot of unnecessarily complicated data structures, which increased the memory requirements and decreased the efficiency of the program. Since C is very rich and powerful language, it can express complex data structure very easily, such as node, edge and the data associated with them. Data associativity appears to be clearer to the user by using C's struct declaration. However, just because of its expressivity and flexibility, it also cause some other negative effects : namely first of all, the use of dynamical space allocation and deallocation makes a program more dynamical and run-time dependent. The explicit pointer use makes the program more flexible, however more complicated. All of these and many other advantages in turn may make it harder for a compiler to recognize the internal data structures, and thus harder for it to utilize available compiler optimization techniques. Therefore C's compiler optimizer may not be as effective as Fortran's. In Fortran, since it is harder to express all these complicated data structures, as a programmer, we have to put more effort to extract the data relationship and try to express them in term of array or array pointers, thus, the data layout in the memory may appear to be more regular than by using individual allocations in C. So the human effort will be trade off in that the compiler optimizer will be more effective. *Figure* 8 shows the speedup with and without the Fortran compiler optimizer. The Fortran compiler optimizer is about twice as effective as the C compiler optimizer(C result is not shown). The C compiler is getting better since more and more people are using C in scientific and engineering computation and people are putting more effort on it. However, we would agree it is very hard for C compiler to completely match Fortran compiler performance since C is just far too flexible and versatile.

In order to improve the performance of the code, we rewrote the program from scratch in Fortran without using DIME, in order to make it easier to parallelize by using Fortran 90, Fortran D, or CM Fortran. The new code was simpler and more specialized, and consequently ran significantly faster than the previous C code on the all major types of RISC processors (*Figure* 9)

However this new code still ran very slowly on certain machines, compared with vendor's specifications (Table 1), especially on the Intel Touchstone Delta, which uses Intel $iPSC860$ processors. For this code we realized only about 1 MFlop on the $iPSC860$, which is theoretically an 40 MFlop processor for 32-bit operations. The random surface program is very floating point intensive, with typical large scale simulations using the equivalent of 1000 hours of a CRAY Y-MP processor [12, 13]. Even so, the results of these simulations are still not conclusive. Speeding up the program is therefore vital to further progress on this problem.

3 Optimization and Memory Hierarchy

Our Fortran code for the random surface simulations ran very slowly on the Intel $iPSC860$, and we were convinced that we could improve the speed of the code substantially for this and other modern RISC processors by some careful optimization.

In the process of optimizing the code, we pursued systematic ways to explore all levels of optimization, namely investigating : 1. possible optimization provided by compiler , 2. basic optimization techniques and 3. more program-specific optimization. The overall aim is to minimize the number of operations and calculations and to best utilize data locality so that maximum available memory bandwidth is utilized, thus the processor computation cycle is not idle.

Step 3. currently has to be done by programmer to explicitly re-arrange the data layout and data associativity so that the processor will retrieve data in cache or main memory in an efficient manner. Hopefully compiler in the future will extract common characteristics from these irregular types of applications and do this step reasonably well.

We believed that the poor performance of the program on the $iPSC860$ processor was probably due mainly to cache misses, since the off chip memory access for this processor takes many more cycles than on-chip cache access, whereas a floating point operation can be done in a single cycle. This is the case for most RISC processors, and can be a problem in utilizing these processors efficiently. Especially iPSC860 on-chip cache is very small, data hit ratio is even more critical to iPSC860 relatively fast processor and directly related to the utilization of cache, since other major types of processors have relatively bigger data cache and instruction cache. We therefore attempted to understand why the program ran so slowly, and tried to optimize the program for RISC processors such as the $iPSC860$, the RS/6000 and others by investigating possible problems, in particular memory access.

Note that this is also an essential step in optimizing parallel programs, since we want to distribute data over processors in such a way that the memory accessed by any processor is as local as possible, to minimize communication costs between processors. On a sequential machine we have a similar kind of memory hierarchy (*Figure* 10), with on-chip cache memory corresponding to on-processor memory for a parallel machine, and off-chip cache memory corresponding to memory on a different processor, which requires extra time to access.

Generally speaking, the sequences of a program execution can be represented as a sequence of memory accesses. The memory study has shown [1] that in the time domain, a process has a tendency in near future to refer to the memory address which were accessed in the recent past; and in spatial domain, the process likely refers to a portion of the memory address physically in the neighborhood of the address which were referred last time.

All these localities influence an efficient usage of hierarchy memory and determine the size of the block to be transferred between memory levels. The replacement policies in many cache and memory design are based on the above two facts.

Based on the idea of memory having a hierarchy structure, and the particular characteristics of certain RISC chips, we can outline a fairly systematic method for optimizing the program. These methods are quite general, and can be applied to any numerically intensive program run on a fast RISC processor. We have benchmarked the results of these optimization techniques on the Intel $iPSC860$, IBM RS/6000, HP 9000, DECstation 5000(not very complete yet), and Sun SPARC 1+.

4 Optimizing Compilers and Basic Optimization Techniques

We first investigate what the compiler provides. Optimization flags with compilers provide local and global optimization, and may perform some pipelining. A comparison of the speed of the program before and after compiler optimization for all the different machines is shown in Figure 8.

For the $iPSC860$, we found that the speed is increased by almost a factor of 2 just by using the NOIEEE compiler flag (This varies with the number of nodes). This causes float and double divides, which are otherwise extremely slow since they are done in software, to be done using an inline divide algorithm [16]. This gives substantial speed-up of functions such as division, square root, exponential and arccos which are used a lot in the our program. In a similar vein, the RS/6000 Fortran compiler has a flag (qrndsngl) which needs to be turned on in order to ensure strict adherence to the IEEE arithmetic standard (i.e. IEEE standard arithmetic is *not* the default for this compiler). The strings code runs about 10% slower with this flag set.

We use our own random number generator subroutine many times in the inner loop of the program, and each call to this subroutine has a function call stack overhead.

The IBM and Intel compilers provide an inlining compiler flag, which allow calls to specified functions, or functions of less than a certain size, to be placed into the code (inlined) rather than using a function call. By using the inline operation on the random number generator we get about a 4% speed-up for the $iPSC860$, but no noticeable speed-up for the RS/6000. The same results are obtained by manual inlining. There is a trade-off here, in that bad inline effects may actually decrease the performance of the code, by increasing the code size and thus increasing overhead due to the small size of the instruction cache. Compilers for other machines may do automatic inlining at a certain level of compiler optimization, and one should be careful to check that this does in fact increase and not decrease the performance of the program.

We know that current compiler technology has not yet matured to the point of automatically taking advantage of many of the features of modern RISC chips. Thus the user needs to carefully reconstruct the program and fully expose the availability of pipelining and data locality to the compiler, in order to take maximum advantage of the data cache.

Firstly, we must try to keep the most frequently used variables in the cache, and reuse them as much as possible while they are still there. This may require re-ordering parts of the code (see examples below). We also tried to reduce unnecessarily large array sizes so that all, or at least most, of the working set data is in the cache and most of data available in fewer pages in virtual memory space. Thus after the transient period for loading this data onto the cache, most of the memory access is to fast cache memory. It is even better to keep data which is re-used in registers. This can be done explicitly by the programmer in C by using the *register* variable type. In Fortran we need to allocate such data to temporary locally declared variables, and hope the compiler takes advantage of this.

```
case 1.
    use                              NOT
       a = b(i)                         c = b(i) + d
       c = a + d                        f = sqrt(b(i))
       f = sqrt(a)

case 2.
    use                              NOT
       a = b(i)                         a = b(i)
       c = a + d                        c = a + d
       f = sqrt(a)                      ...
                                        many lines before using 'a' next
                                        ...
                                        f = sqrt(a)
```

Secondly, in trying to optimize code, one usually thinks of an assignment as being fast and a multiplication as being slow, however this is not the case with modern RISC processors. On the $iPSC860$, for example, a load operation takes two clock cycles and a store operation three cycles while add or multiply take just one. We should thus eliminate redundant assignments and minimize memory access, especially external memory access, use constants rather than variables, and try to avoid data conversion.

Thirdly, we try to minimize the number of expensive calculations, such as division, square root, exponential and arccos, which all occur in the strings code. If a calculation is expensive and repeatedly used in several places, it may be possible to calculate it once and assign it to a global variable, so further calculations are reduced to using this "look-up table". We noticed that normal direction and edge action were calculated in the update process, and then used in subroutine measurement, since the computation for these quantities are extremely expensive, we set up global variables for them. Certainly it is a trade-off in terms of program modularity. By doing this, the subroutine measurement got speed-up more than 20%. We also tried to utilize the concurrent add and multiply operations. An addition combined with a multiplication is performed in about the same time as a multiplication on most RISC processors.

Fourthly, branches (IF statements) and loops (DO or FOR statements) can have a large overhead. We should try to avoid IF statement since they slow down any pipelining [2]. Optimizing loops is somewhat problematic. Using the DO loop is a very good way to utilize the instruction cache, since it allows a lot of code to reside in the generally small instruction cache space. In most processors the CPU can take care of the loop while the floating point unit deals with the operations in the loop. However there is still a substantial overhead in using a DO loop, so we should merge DO loops as much as possible, and in cases where the loop contains very few instructions it should be "unrolled". This means that if the number of iterations of the loop is very small (for example, a loop over 3 dimensions), the loop should be eliminated and the instructions written out explicitly for each iteration (each dimension). If the number of iterations of the loop is large, say 100 iterations of a single line of code, then it should be unrolled into a loop over say 5 iterations of 20 lines of code. The optimal number of lines of code in the loop after such a split should be such that the loop just fits in the instruction cache. unrolling of loops should be done by the compiler, but this is not always the case.

Also, since data is usually loaded into the cache a few words at a time, we should try to keep the loop stride smaller than the length of this cache line to avoid possible "thrashing" of the cache (continual reading and writing of data to and from cache). This would also increase the chance of pipelining multiple read and write operations since the working data will be in same cache line or nearby.

Another crucial point to note for multiple loops over elements of arrays of more than one dimension is that the loops must be ordered in such a way that the inner loop corresponds to the array index which changes fastest. In Fortran this is the first index, while in C it is the last index, so optimal loop ordering will depend on the language used.

5 Program-Specific User Optimization

After implementing the generic optimization techniques given above, we turned to more problem-specific methods to speed up the code. Here we are still interested in minimizing memory accesses, maximizing cache usage and minimizing the number of operations. Now though we are aiming to do this by studying the specifics of our particular problem, and finding ways to change the algorithm so as to better match the data structures and data access to the hierarchical cache memory of the processor. The calculations of quantities associated with a node requires one to know the nearest neighbor and the next nearest neighbor information, it takes lots of operations and data access in the non optimized code just to figure out the relative position between neighbor links and points since the mesh is irregular. And even worse since there are many conditional branched IFs and GOTOs involved, we could expect that there are lots of memory jumps and instruction would not be efficiently pipelined. There are many of these types of problem in the unoptimized code. By appropriately ordering the operations at each mesh point, and in particular the traversal of the mesh data structure, so as to maximize data locality, we realized quite substantial gains in performance.

Here we show a typical example. Initially, each node in the mesh has six nearest neighbor nodes, so it has six connecting links. Each link has a pointer *node_of_link* to point to its node and a pointer *link_next_link* to point to its neighbor links. Initially everything is in order (*Figure* 11). As the mesh is changed during the simulation by flipping the links, the associativity between nodes and links and neighboring links changes. Links pointing to the same node may now not be located in contiguous sections of the memory, and the link pointer may not point to the nearest neighbor link among the links around a node (*Figure* 12). That is the case in the original version of the code, for which the links around a node are not ordered after flipping. However, the calculation of physical quantities associated with each link requires the information at the nearest neighbor links around their boundary nodes. Since the links are not ordered, to figure out the left side link and right side link of a given link, the original code first looks for the left side and right side nodes S_1 and S_2 (*Figure* 7), and then makes a comparison to see which link connects the node pair (S_0,S_1) and (S_0,S_2).

```
      ...
      s1 = node_left_to_link(this_link)
      s2 = node_right_to_link(this_link)
 111  temp_link = start_link_of_node(s0)
 112  if (node_of_link(temp_link) .eq. s2) then
          s0s1 = temp_link
          goto 113
      else
          temp_link = link_next_link(temp_link)
```

```
            goto 112
         end if
 113     continue
         ...
```

Here we have changed the algorithm to improve data locality. At the time we flip the link, we re-order the links immediately, each link getting a pointer to its nearest neighbor link (*Figure* 13). So we get its nearest neighbor links (S_0S_1, S_0S_2, etc.) basically for free (*Figure* 7).

```
    ...
    s0s1 = link_next_link(this_link)
    s0s2 = link_last_link(this_link)
    ...
```

By implementing this and making other similar changes, we achieved the biggest improvement in speed. It eliminates a lot of IF statements, pointer chain searching operations, and several unnecessary large arrays. It also provides much useful information for other calculations at no extra cost.

In general, it is much harder to exploit data locality in a dynamic mesh than a fixed mesh. Because the neighbors of nodes and links and their corresponding ordering are changing as the simulation continues, the neighboring physical memory locations may not reflect the neighbor of a link or a node, and vice versa (Fig. 4). The optimization may require a run-time preprocessing strategy to predict more precisely the data locality.

6 Performance Comparisons

We have compared the performance of this code and the effect of these optimization techniques on a number of current RISC-based processors and workstations, in particular the Intel *iPSC*860, IBM RS/6000, HP 9000, DECstation 5000 (data for DECstation is not very complete), and Sun SPARC 1+.

Figure 8 shows effects of the compiler optimization along on these processors. HP's, IBM's and iPSC 860's Fortran compiler optimizations yield very big improvements in the performance of our program, by about a factor of from 2 to 2.5. DEC's amd Sun's are relatively poor.

Figure 14, from chart a to chart d, show the absolute time elapsed for one call to subroutine *update-nodes* and *update-links* respectively on different processors, we put the time for optimized and unoptimized code in the same graph.

Figure 15, chart a and chart b, show the speed-up of subroutine *update-links* and subroutine *update-nodes* respectively. At this time, we put the data for different types of machines on one graph for easy comparison.

We can see that manual optimization of *update-nodes* does not give big the speed-up for most processors except iPSC 860. *update-nodes* as explained previously deals with a fixed connectivity mesh. The data access appears to be smoother than in *update-links* . So the major optimization applied on this subroutine are basic optimization technique in our paper section 4. These techniques are the main contribution to the speed-up. Since the iPSC has a relatively small data cache, it seems that these re-arrangement are quite crucial to speed. The speed-up is about a factor of 1.8. The HP data shows that this has a small effect since the HP has a relatively large data cache, and the mesh we used is small, so that the optimization effect is not large enough to appear. However, we would expect as the size of mesh increased, the speed-up would be noticeable.

update-links shows about a factor of 1.5 speed-up on the HP and the IBM, and about a factor of 2.0 on iPSC860. As what we explained earlier, most of this improvement comes from the program-specific user optimization.

Finally, *Figure* 16 shows the overall performance of the unoptimized code and optimized code in Mflops on different processors.

7 Conclusions

From the study of optimizing this real application of irregular dynamical mesh problem, we have learned that we need to be careful to construct the code and the data structures so that they make most efficient use of the memory hierarchy characteristics and pipelining of modern RISC processors. Current compilers can provide good optimization, but do not recognize and exploit all the information on data locality and pipelining opportunities. Substantial gains in performance can be achieved if the user is prepared to restructure the code to help provide the compiler with this information, especially programs with adaptive, irregular data structures such as the random surfaces application.

The interesting and attractive parts of the random surfaces simulation are that one code has two major different aspects : subroutine *update-nodes* is irregular but static in terms of data association and data distribution; however, subroutine *update-links* is dynamic as well as irregular. Both features are very useful to characterize the behaviors of RISC processors from different angles. That is why we use it as one of important benchmark programs.

Our results show that the dynamical part of the program needs to have more aggressive code re-arrangement or even algorithm modification in order to achieve a relatively big performance improvement. Simply using the basic general optimization techniques does not have real effect on the dynamical part although it does speed up the static part. That means the compiler needs to be more intelligent so that it can look deeper into the program and extract more information about data association if the compiler wants to do this level optimization. That is one issue that High Performance Fortran will eventually address. Dynamical run-time techniques have to be used to explore more precisely the behavior of data locality. we are trying to extract more general strategies from ours and other similar type of irregular applications to support the High Performance Fortran development. This study is a pre-investigation for the on-going development of the High Performance Fortran compiler by Syracuse University, Rice University and University of Maryland.

We have obtained a first-hand data that tells which portion of the techniques used are most significant to the future High Performance Fortran compiler development. We have systematically shown the approaches to improve the performance on almost all major types of RISC processors commonly used. As we understand more about irregular data structure, we will write more efficient parallel codes on the CM5 and the Touch Stone Delta. We expect that these data locality techniques can be used by data parallel compilers such as those for High Performance Fortran to allow these languages to outperform conventional Fortran even on sequential processors. Thus the High Performance Fortran not only achieves the parallelism but also exploits the capacity of each individual processor, especially for very irregular problems (such as *update-links*).

8 Acknowledgments

We would like thank NPAC and CRPC computing facilities. We gratefully acknowledge discussions and help from Enzo Marinari who wrote the initial Fortran code, Mark Bowick, Paul Coddington, Geoffrey Harris and Geoffrey Fox.

References

[1] H. S. Stone High-Performance Computer Architecture (Addison-Wesley Publishing)

[2] W. Stallings, "Reduced Instruction Set of Computer Architecture", Proceedings of The IEEE, **Vol 76**, No.1, (Jan. 1988) 38

[3] G. C. Fox, *et al.*, Solving Problems on Concurrent Processors, Vol. 1 (Englewood Cliffs : Prentice-Hall, 1988)

[4] R. Das, D.J. Marvriplis, J. Saltz, S. Gupta and R. Ponnusamy, "The Design and Implementation of a Parallel Unstructured Euler Sover Using Software Primitives",ICASE Report No. 92-12

[5] R.D. Williams, in Proc. of the 3rd Hypercube Conference, Pasadena, 1988, ed. G.C. Fox, (ACM Press, New York, 1988).

[6] "High Performance Fortran Report", Syracuse University Internal Report

[7] F. David, in *Two Dimensional Quantum Gravity and Random Surfaces*, eds. D.J. Gross, T. Piran, and S. Weinberg, (World Scientific, Singapore, 1992).

[8] V. A. Kazakov, *Phys. Lett.* **150B**, 282 (1985); F. David, *Nucl. Phys.* **B257**, 45 (1985); J. Ambjørn, B. Durhuus and J. Frölich, *Nucl. Phys.* **B257**, 433 (1985).

[9] *Statistical Mechanics of Membranes and Surfaces*, eds. D. Nelson, T. Piran, and S. Weinberg, (World Scientific, Singapore, 1989).

[10] S. Catterall, *Phys. Lett.* **220B**, 207 (1989).

[11] V. A. Kazakov and A. A. Migdal, Nucl. Phys. **B311** (1988/89) 171.

[12] J. Ambjørn, J. Jurkiewicz, S. Varsted, A. Irbäck and B. Petersson, *Phys. Lett.* **275B**, 295 (1992); J. Ambjørn, A. Irbäck, J. Jurkiewicz and B. Petersson, "The Theory of Dynamical Random Surfaces with Extrinsic Curvature", Niels Bohr Institute Preprint NBI-HE-92-40.

[13] M. Bowick, P. Coddington, L. Han, G. Harris and E. Marinari, "The Phase Diagram of Fluid Random Surfaces with Extrinsic Curvature", Syracuse preprint SCCS-357, submitted to *Nucl. Phys.* **B**.

[14] C.F. Baillie, D.A. Johnston and R.D. Williams, *Comput. Phys. Commun.* **58**, 105 (1990).

[15] C. Baillie, D. Johnston and R. Williams, *Nucl. Phys.* **B335**, 469 (1990).

[16] Intel i860 Microprocessor Family Programmer's Reference Manual