Frontiers '95 (Proceedins of the Fifth Symposium on The Frontiers of Massively Parallel Computing)
MacLean, VA (Feb 6–9), 1995, pp. 222–230.

Optimizing Irregular Computations on SIMD Machines: A Case Study

John S. Conery,* Michael Lynch,† and Tommy Hovland*

* Department of Computer and Information Science
† Ecology and Evolution Program, Department of Biology

University of Oregon
Eugene, OR 97403

email: conery@cs.uoregon.edu

Abstract

Data-parallel computations with regular structure — fixed data size and predictable control patterns — can be implemented efficiently on SIMD architectures. However many large applications have irregular structure, either data sets that vary in size as the computation progresses or control structures that select different subsets of the processors at each stage of the computation. In this paper we describe a stochastic biology simulation and some of the methods we used to improve its performance on the MasPar MP-1104. We present a simple model for evaluating the performance of a data parallel application and use the model to improve the performance of the simulator.

Keywords: data parallelism, performance models, source level optimization, stochastic simulation, computational ecology

1. Introduction

Scientific applications with uniform structure map very well to SIMD architectures. Examples of such applications include seismic modeling with a finite grid method [8], circulation patterns in the ocean and atmosphere [5], and molecular dynamics [4]. These programs typically have fixed size data and calculations that are uniform over the entire data set.

Many important scientific problems are not as regular. The data may vary in size during the computation, or, as in the case of a parallel search, data structures may not be known in advance and must be mapped to the architecture as they are created [6]. In addition, the calculations done at each data point may be different, which means subsets of processors are idle a significant amount of time.

In this paper we describe our experience with optimizing an irregular data-parallel application. The program is a stochastic simulation of the buildup of mutations that lead to the extinction of endangered species. It is a natural data-parallel application since the same calculation, for example combining the genes from individuals in the current generation to produce offspring for the next generation, is done repeatedly over very large data sets. Because the simulation is stochastic the program is also highly irregular: some populations go extinct before others, and individuals that survive to form the next generation must be evenly redistributed among the processors simulating that population.

After a brief overview of the simulator to provide background information, we present a simple performance model that characterizes the efficiency of a data-parallel computation. We then show how the model can be used to evaluate parts of the simulator and suggest changes in the source program that improve efficiency and as a result decrease the overall execution time. We also use the performance model to analyze a trade-off between communication and computation in the load balancing step that distributes individuals to processors.

2. The Mutational Meltdown Simulator

One of the factors that may contribute to the extinction of small populations is the buildup of harmful mutations. Field data collected from a variety of species indicates that on average new offspring have one new mutation. In a finite environment (on an island, for example), mutations that are passed on to subsequent generations may eventually become *fixed*, that is they are present in every individual in the population. The phrase "mutational meltdown" refers to the fact that at some critical point there will be a

sufficient number of fixed mutations to guarantee extinction within a few generations.

Simulations show the population goes through three phases: a short initial phase in which new mutations are introduced, a prolonged intermediate phase in which most new mutations drop out but others become fixed at a steady rate, and a short final meltdown phase in which the population is quickly driven to extinction as a result of too many mutations [7]. What this simulation shows is that mutations alone are enough to cause the extinction of small, fixed-size populations even if habitat and other environmental conditions remain unchanged.

In order to measure the effect of mutation rate, initial population size, and other parameters on the rate of extinction we developed a parallel simulator to run on MasPar SIMD machines. In the simulator an individual is represented by a string of bits: a 1 at a particular site (or *locus*) means the individual has a mutated gene at that locus. Initially all loci are set to 0. The relative health of an individual is a function of the total number of mutations it has accumulated.

The main loop of the simulator creates a new generation from the current generation. We randomly select two individuals, combine their genes to form the representation of a new individual, add a random number of new mutations, compute the health of the new individual, and, if it survives, add it to the next generation. The most important parameters that control the simulation are:

- The carrying capacity, K. This is the maximum number of individuals the environment can support. Values in our simulations range from 2 to 512.
- The reproductive rate, R. This is the average number of offspring each individual will create. Typical values are 2, for species such as mammals, to 10,000, e.g. for plants with a large number of seeds.
- The mutation rate, μ. This is the expected number of new mutations in each new individual.

When building a new generation the simulator generates $n \times R$ new offspring, where n is the size of the current generation. The first K to survive become the next generation. Eventually none of the offspring survive and the population is extinct.

On a MasPar system we use a fixed number of PEs, known as a *PE group*, for each population. The size of the group determines how many individuals are stored on each PE. For example, with K = 64 and a group size of 16 there will be 4 individuals on each PE. The PEs in a group are contiguous to allow scan operations to count survivors and perform other population-wide operations in log time.

The simulation experiments can be very time-consuming. For each combination of parameters is it necessary to simulate from 250 to 500 populations in order to measure the mean and standard deviation of the number of generations to extinction. At the time we first started using the MasPar, simulations for asexual reproduction, which are much simpler than the simulations described here, were taking over a week (wall clock time) on a SPARC-2 workstation. The first parallel simulations ran on an MP-1101 about 15 times faster (CPU time) than the sequential version. The latest versions of the program simulate sexual reproduction and handle populations as large as 512 individuals. The largest simulation ran for several CPU-days on an MP-2216. It simulated 60,000 generations of 64 populations of size 128, creating over 10^9 individuals.

Most steps in the simulation map very well to a SIMD architecture. For example, the main loop is easily parallelized: each PE in a group can select two members of the current generation, perform the steps necessary to generate a new individual, add a random number of new mutations, count the mutations in the new individual, and determine whether or not it survives. The sources of irregularity are also readily apparent. For example, at the end of the loop that creates the new generation the survivors will be scattered among PEs within the group, and to make the creation of the following generation more efficient they need to be redistributed evenly.

3. Performance Model for Irregular Data-Parallel Computations

In a data-parallel computation a processing element (PE) is either idle or it executes the instruction broadcast by the control processor. The set of all PEs that execute a given instruction is known as the *active set*. The *efficiency* of a computation is defined by the relative amount of time processors are active. In a perfect program, all processors are active at every step and the efficiency is 1.0. Efficiency is important in program optimization because calculations often require a fixed number of steps. If they can be reordered so the active set is larger in key regions of the program and more steps are executed in parallel, then the overall execution time will be reduced.

The efficiency of a data-parallel computation is expressed in terms of the size of the active set:

$$E = \left(\sum_i p_i \cdot t_i \right) / T$$

Here p_i is the size of the active set during program interval i, expressed as the percentage of processors that are active; t_i is the duration of interval i; and T is the total execution time. Note that if all processors are busy during every interval (p_i = 1.0 for all i) the efficiency is 1.0 since the sum of the lengths of each program interval should equal the

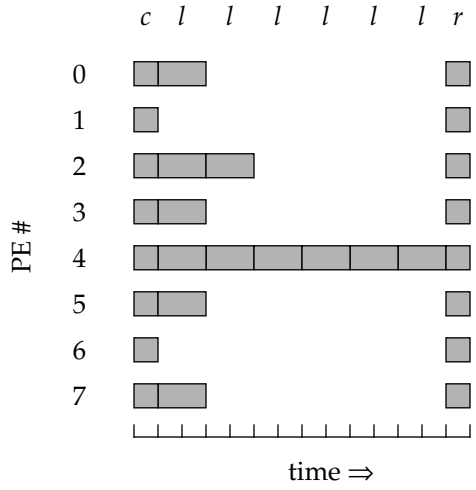

Figure 1. Poisson random number generator: Gantt chart for a call to `p_poisson(1.0)` on eight PEs and table of probabilities for $\lambda = 1.0$.

n	p(x=n)	p(x≥n)
0	0.367879	1.000000
1	0.367879	0.632120
2	0.183940	0.264241
3	0.061313	0.080301
4	0.015328	0.018988
5	0.003066	0.003660
6	0.000511	0.000594
7	0.000073	0.000083
8	0.000009	0.000010
9	0.000001	0.000001

total execution time. There are several other ways of describing efficiency (e.g. [11]) but we prefer this description because it makes explicit the role of the active set and the fact that it may change in size from one program interval to the next.

As an example, consider a Poisson random number generator, which is a function that returns an integer value distributed around a mean (called λ). A common implementation is based on a simple loop that draws a uniform random number between 0 and 1 and performs one multiplication and one comparison [10]. The return value is the number of loop iterations.

In a data-parallel implementation we want each PE to compute an independent random deviate. The straightforward implementation uses a parallel `while` loop: each PE independently draws its own random number and does a local multiplication and comparison. As each PE exits the loop the active set becomes smaller. During the i^{th} iteration the PEs that are still active are those that will return a value of i or greater. The probability that a PE will return a value greater than or equal to n, denoted $p(x \geq n)$, gives us the size of the active set on iteration i. Let t be the time required for one loop iteration. The efficiency of a parallel Poisson generator is then

$$E = \left(\sum_{i=1}^{m} p(x \geq i) \cdot t \right) / mt \approx 1/m$$

where m is the largest deviate computed on any PE.

According to the table in Figure 1, for $\lambda = 1$ the probability of generating a 6 is .0006, or $1/1667$. In other words, in 1667 calls to a Poisson generator we are likely to see one 6. Thus on an MP 1104, with 4096 processors, it is highly likely that one PE will execute the loop 6 times, and the expected efficiency for the body of the Poisson function would be $1/6 = 0.17$. It is interesting to note that the efficiency of the Poisson function will decrease with increasing machine size since there is a higher probability that one of the PEs will generate a large deviate.

As another example, the Gantt chart in Figure 1 shows eight PEs executing a call to the Poisson function, including the time to set up the loop (labeled *c* at the top of the chart), six loop iterations (labeled *l*), and the program interval where the return value is passed back (labeled *r*). Including call and return intervals executed by each PE the efficiency in this example is

$$E = (1.0 \times 1 + 0.75 \times 2 + 0.25 \times 2 + 4(0.125 \times 2) + 1.0 \times 1)/14 = 0.357$$

Without the call and return intervals, during which every processor is active, the efficiency is 0.25, which is closer to the theoretical value of .17.

Because the Poisson random number generator is called from the inner loop of the simulator — it determines the number of mutations to add to offspring — we experimented with a version of the simulator that precomputes a table of Poisson deviates. When the simulator is initialized each PE builds a table with several thousand deviates. At runtime, each PE draws a random element from the table, which is a constant time operation on each PE. In one simulation that involved the generation of over 100,000,000 individuals the total execution time on an MP-1104 decreased about 25%, from 1099 seconds to 832 seconds.

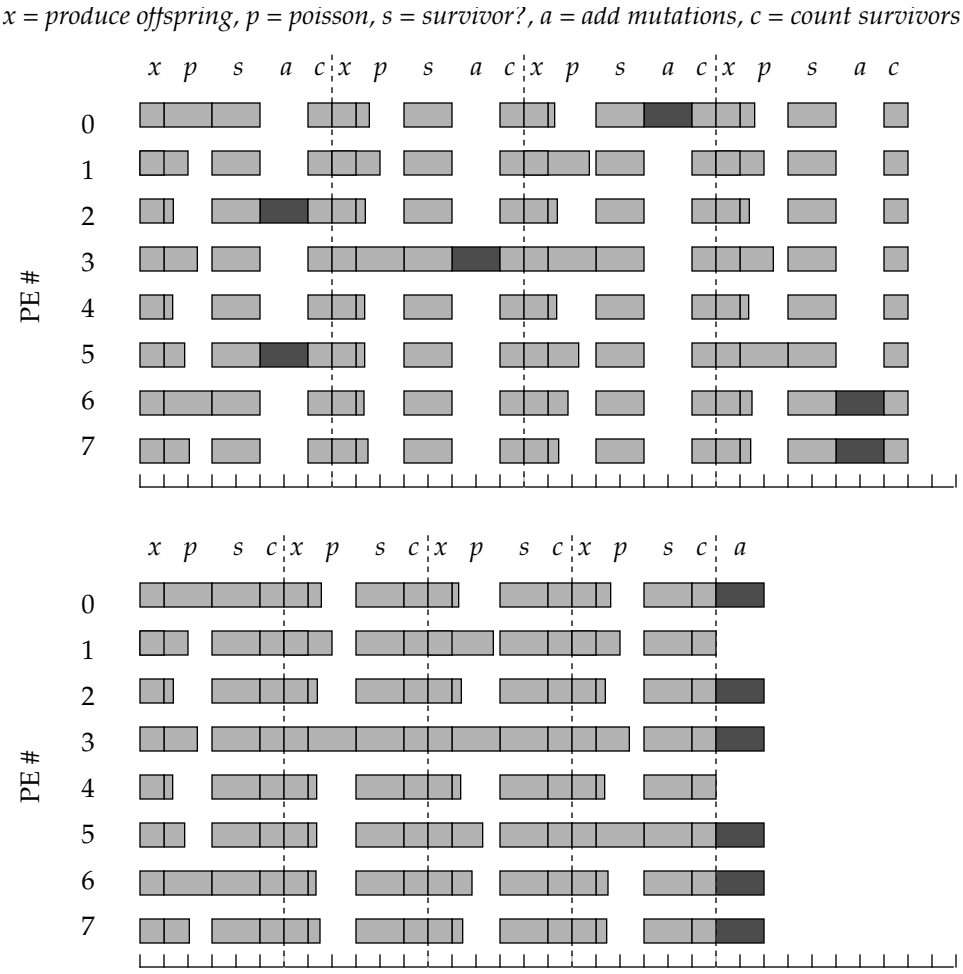

Figure 2. Gantt charts for original loop (top) and two loops after fission (bottom).

4. Loop Fission

The heart of the mutational meltdown simulator is the procedure that builds the next generation:

```
n = nsur = 0;
for (i = 0; i < limit, nsur < k; i++) {
   produce_offspring();
   nm = p_poisson();
   if (survivor(nm)) {
      add_mutations(nm,n);
      n += 1;
   }
   nsur = count_survivors(n);
}
```

The parallel `for` loop iterates until the number of survivors in the new generation (`nsur`) reaches the desired population size (`k`) or until the maximum number of offspring have been generated. The latter test is needed because in the final meltdown phase of the simulation there will be so many mutations in the population that fewer than `k` individuals will survive.

`produce_offspring` is executed in parallel on each PE to generate a new individual. If with `nm` new mutations the new offspring would survive the mutations are inserted into the representation of the new individual by the call to `add_mutations`. Finally, `count_survivors` sums `n`, the number of survivors on each PE, to compute the total number of survivors in the population.

In this section we describe a source level transformation, commonly known as *loop fission* [12], that improves the efficiency of this piece of code. The basic idea is to break the single `for` loop into two independent loops. This transformation is often counterproductive in sequential programs that are optimized by combining loops to reduce loop overhead, but in a data-parallel program performance

is improved if the operations within the loop that are executed by a relatively small active set are postponed and executed in a separate loop in a manner that increases the size of the active set.

The call to `add_mutations` is the key to the performance of this part of the simulator. These calls are highlighted in the Gantt chart in Figure 2. Note that in each loop iteration the number of PEs that executes the call to `add_mutations` is fairly small. The size of the active set depends on the probability that a PE creates an individual that will survive to the next generation. Early in the simulation, when all offspring are healthy, this probability is fairly high. But as the mutations build up in the population the probability that any one individual will survive drops and more loop iterations are required to construct the new generation. Early in the middle phase of the simulation the system is executing the `for` loop the maximum number of times.

The size of the active set in the call to `add_mutations` depends on three parameters: K, the population size; R, the reproductive rate; and G, the PE group size, which is the number of PEs dedicated to each population. To build a new generation the simulator is required to create up to $K \times R$ new offspring. The maximum number of loop iterations is $(K \times R)/G$, i.e. the invocations of the loop body are spread evenly among the G PEs. If the program takes the maximum number of loop iterations to generate the K survivors that will form the new generation, on average the number of survivors per loop iteration is $K/((K \times R)/G) = G/R$. The percentage of PEs that are active during calls to `add_mutations` is the number of survivors per iteration divided by the group size:

$$p = \frac{(G/R)}{G} = \frac{1}{R}$$

Thus for higher values of R we expect to see worse performance in the calls to `add_mutations`.

As a concrete example, with $K = 32$ and $R = 100$ the simulator will create up to 3200 offspring as potential members of the next generation. Suppose the group size is 16. That means there will be up to 200 iterations of the loop (so 16 processors can execute the loop body a total of 3200 times), out of which will emerge 32 survivors. In the middle phase of the simulation, when all loop iterations are required to build the new generation, the average number of survivors per iteration is $n = 32/200 = 0.16$. The average size of the active set is the number of survivors per iteration divided by the number of PEs in the group: $n/16 = 0.01$, which is $1/R$.

Since `add_mutations` is nontrivial — it needs to coordinate with a garbage collection procedure to find free loci in which to insert the mutations — we need to maximize the size of the active set when it is called. The solution is to transform the program so that instead of one loop there are two. During the first loop we save the number of mutations that should be added to each individual, and in the second loop we iterate over all local survivors and call `add_mutations`:

```
n = nsur = 0;
for (i = 0; i < limit, nsur < k; i++) {
   produce_offspring();
   nm = p_poisson();
   if (survivor(nm))
      mc[n++] = nm;
   nsur = count_survivors(n);
}
for (i=0; i < n; i++)
   add_mutations(mc[i],i);
```

In this version `mc` is an array on each PE that keeps track of the mutation count for each local survivor. The Gantt chart for the new, optimized version is shown below the chart for the original in Figure 2. The calls to `add_mutations` in the first loop have been replaced by assignment statements (which are instantaneous on this time scale) and one call to `add_mutations` in the second loop is sufficient to insert mutations into all survivors.

Table 1 presents data that verifies the performance improvement as a result of moving the call to `add_mutations` to a second loop. As was the case with the Poisson generator, the optimization pays increasing dividends as the length of the simulation, which is proportional to $K \times R$, increases. As expected the improvement is most pronounced for the smaller values of $1/R$, i.e. when the fewest processors are in the active set in the calls to `add_mutations` in the original `for` loop.

5. Load Balancing Interval: Tradeoff Between Control and Communication

The communication channel used for point-to-point communication in the MasPar is known as the router. It uses three levels of cross-bar switches to connect a PE to any other PE [2][9].

Table 2 shows the performance of the router using the MPL primitive `rfetch`. To collect this data we controlled the size of the active set by selecting a different subset of the PEs to execute the `rfetch` procedure. The PEs in the active set were not selected randomly; instead they were distributed evenly across the machine, which had the effect of distributing the contention for the first level of cross-bar switches in the router. Each active PE selected another PE at random from which to fetch a specified amount of information. The "message size" is listed in the left column of

Table 1: Improvement as a Result of Loop Fission Optimization

K	R	$K \times R$	$p = 1/R$	Execution Time (sec)		Improvement
				Original	Optimized	
2	2	4	.50	8.61	8.54	1.1%
8	2	16	.50	14.06	13.53	3.8%
16	2	32	.50	17.78	16.68	6.1%
32	2	64	.50	31.65	29.15	7.9%
2	100	200	.01	54.07	53.22	1.6%
8	100	800	.01	105.50	98.21	6.9%
16	100	1600	.01	275.41	245.89	10.7%
32	100	3200	.01	643.27	466.16	27.5%

the table, and the percentage of PEs that were active is given in the top row.

The left half of Table 2 shows the latency, or amount of time required to transfer the data. As expected the time goes up almost linearly with the message size. The correspondence is not exactly linear with a slope of 1.0 because the connection pattern was random and timings may vary because of higher or lower collision rates. Note also that the time increases linearly as more PEs are active, which is also to be expected since the number of active PEs affects the contention in the cross-bar switches. Note that even though latency is worse with increasing message size and active set size, the bandwidth, shown in the right half of the table, improves with larger messages and to a lesser degree with increasing active set size.

Given the data on communication latency it is apparent that one way to improve overall performance would be to reduce the size of the active set during router operations. With fewer PEs using the router the length of a program interval that calls router procedures will be shorter. However this approach is in conflict with the program transformation described in the previous section, which has the goal of *maximizing* the size of the active set. In this section we describe a part of the mutational meltdown simulator where we were faced with a trade-off between these two conflicting goals.

Recall from the previous section that a key step in the construction of the next generation is a procedure that sums the number of survivors across all PEs in the group in order to compute the total size of the new generation. This procedure is also responsible for load balancing, to make sure that the new individuals are spread evenly among the PEs. For example, with a population size of 48 and a group size of 8 there will be 6 individuals on each PE. In building up the new generation the survival of any individual is a random event, and thus it is possible that by the time 48 new individuals have been generated they will not be distributed evenly within the group. Since later steps will be much more efficient if all PEs have the same number of individuals — a fact predicted by our performance model and verified by experiments — it is important to use a load balancing algorithm to distribute survivors in the new generation.

The load balancing method we use is known as *scan-directed load balancing* [1]. Periodically we check to see if any PEs have excess individuals and if other PEs have open slots for individuals; this check is implemented by two scan operations (parallel prefix sums) which take time proportional to $\log_2$ of the group size. If the check determines that it is time to redistribute the new generation, the data transfer operation is invoked. It uses the router to move data in parallel from PEs with excess individuals to PEs that have open slots for individuals.

The performance trade-off is in the fact that the communication step will be more efficient if we do load balancing after every iteration of the loop that creates new individuals, since fewer individuals will be transferred, but the control overhead will be minimized if we perform the check less often and allow more excess individuals to build up on the PEs. For example, assume we always transfer individuals as soon as the check procedure detects an imbalance. If we check with period $P = 1$, i.e. after each call to the procedure that creates new individuals, then the odds are that few individuals will be transferred. On the other hand, suppose there is room on each PE to buffer four extra individuals. Then it is possible to do a check every

Table 2: Router Performance on Random Communication

% Active PEs / # bytes transferred	Latency (sec)					Bandwidth (MBytes/sec)				
	20%	40%	60%	80%	100%	20%	40%	60%	80%	100%
16	0.001	0.001	0.002	0.004	0.002	17.8	16.9	18.1	13.6	25.2
64	0.002	0.003	0.005	0.006	0.007	22.9	30.0	31.1	32.6	36.5
256	0.007	0.012	0.017	0.023	0.026	29.4	34.9	37.2	36.4	40.2
1024	0.024	0.044	0.073	0.097	0.114	34.3	38.3	34.6	34.6	36.8
4096	0.096	0.193	0.270	0.366	0.414	34.8	34.8	37.3	36.7	40.5

fourth time we create new offspring ($P = 4$), which reduces the amount of time spent in the check procedure. However, the number of PEs with excess individuals will now be higher, the active set in router operations will be larger, and the router operation itself will take longer.

Gantt charts that illustrate the two alternatives are shown in Figure 3. In the chart on the left, the program checks for load imbalance after every iteration of the loop that generates new offspring. Few PEs have excess individuals or empty slots, so the length of the data transfer steps (highlighted) are relatively short. In the chart on the right the program checks every fourth iteration. Now more PEs participate in the data transfer and the communication step is longer. Note that in general the total time spent in transferring data should remain constant — there are fewer communication steps but those steps are now longer because more PEs are active — but the overall time should be lower because there are fewer calls to the procedure that checks for imbalance.

In the mutational meltdown simulator, the analysis of the average number of survivors per iteration makes it clear there is little to be gained by minimizing the size of the active set for router operations during load balancing. With an average of $1/R$ survivors per iteration the active set will already be very small and there will be very few survivors moved by load balancing every iteration. The data in Table 3 confirms this prediction. The active set size is close to $1/R$ when $P = 1$ and changes very little with increasing P, while the time spent checking for imbalance does go down with increasing P. Since the active set size was so small to begin with there is little change with $P = 4$ and there is no loss in performance from increased traffic on the router with higher P.

The observation that there are so few survivors per loop iteration suggests we might be able to lengthen the period and make p a function of both R and the size of the buffer. To be safe we will need a two-stage check for imbalance: an initial, inexpensive check to see if any PE has created enough survivors to fill its buffer, plus a more complete check that totals the number of excess individuals and empty slots across all PEs. This modification is the subject of a current project

6. Summary and Discussion

Describing the performance of parallel program in terms of the percentage of processors that are active is not new; for example, Stone uses figures similar to the Gantt charts in this paper do describe the efficiency of array processors in a discussion of the inherent limits of speedups of programs with sequential components [11]. In this paper we presented the formula in a form that emphasizes the relationship between the length of a program region and the size of the active set for that region to show how the size of the active set contributes to overall program performance.

We used this performance model to analyze a critical part of an ecology simulation. This analysis lead to a “loop fission” transformation that moved a key procedure call out of one loop and into a separate loop. The use of separate loops is more efficient because the active set is larger when the operation is called in the second loop; in the original version only a small percentage of processors were active when the procedure was called.

We also analyzed the active set size for a part of the simulator that requires a load balancing operation. On the surface it appeared there would be conflicting goals in this portion of the simulator: on the one hand router performance improves if the active set is small, since there will

Key: p = produce offspring, c = check for imbalance, x = transfer excess individuals

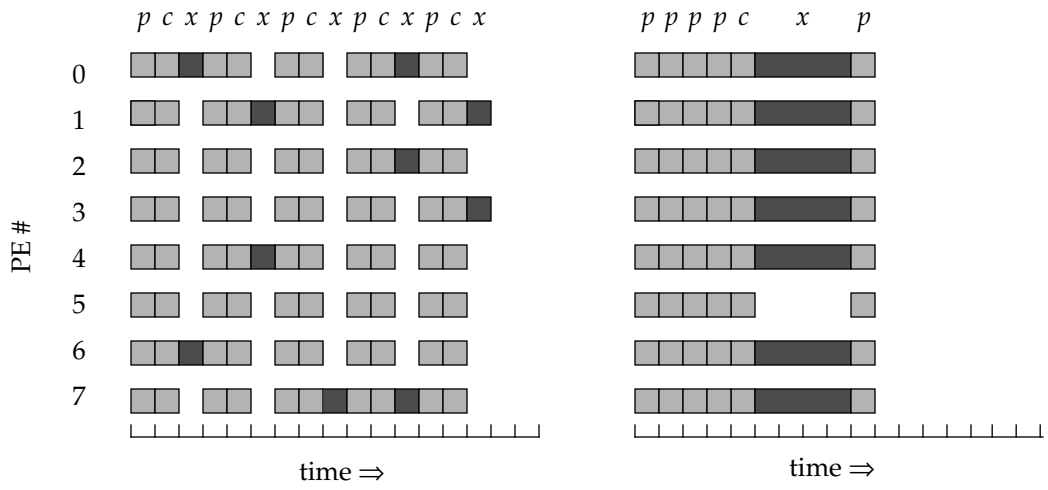

Figure 3. Checking for load imbalance. Left: P = 1; Right: P = 4.

Table 3: Check Period vs. Active Set Size in Load Balancing

		$P = 1$	$P = 2$	$P = 4$
$R = 2$	#checks	184	146	73
	check time	0.107	0.078	0.033
	xfer time	0.336	0.492	0.455
	active set	.050	.060	.105
$R = 10$	#checks	1296	735	433
	check time	0.828	0.430	0.214
	xfer time	2.284	2.292	2.107
	active set	.040	.047	.057
$R = 100$	#checks	11482	5453	3815
	check time	8.549	3.992	2.667
	xfer time	11.788	8.566	8.491
	active set	.013	.023	.030

be less contention, but on the other hand, according to the performance model, it is best to have large active sets for key operations. By looking closer at this situation, however, it became clear that the active set was already very small during load balancing steps and that the best strategy for this part of the simulator was to postpone communication as long as possible.

While decreasing the size of the active set during router operations did not pay dividends in this application, it may be worth considering in other applications, particularly if the communication pattern is not random. If there are "hot spots" in the PE array that are sources or destinations of information, contention in the router will slow down communication. In extreme cases, for example where each PE needs to fetch 4KB from just one PE, it takes the router over 20 seconds to transmit the data to all 4K processors of an MP 1104. Applications where hot spots are unavoidable should consider source transformations that decrease the size of the active set for router operations.

The loop fission transformation has proven to be useful in other applications we have developed, including one that would fall into the category of a highly regular SIMD application. An MPL program that calculates the potential energy in DNA molecules uses a 1-to-1 mapping of atoms to PEs [3]. A loop that sends atom descriptions around a virtual ring in order to build information about bonds between atoms has very much the same structure as the loop in the ecology simulation that builds the next generation: at each iteration only a small subset of the PEs will be active since only a few atoms form bonds with the atom that is currently passing by on the virtual ring. Operations that were inside the loop became much more efficient when they were moved to a second loop.

Descriptions of both of our programs — the mutational meltdown simulator and the potential energy calculation — will be part of a new public domain textbook developed by the Computational Science Education Project (CSEP). CSEP maintains a hypertext server at Oak Ridge National Labs. Readers will be able to use Mosaic or other WWW browsers to read descriptions of the programs, download the code, or even execute the code remotely and download the results. The URL for the CSEP project is `http://csep1.phy.ornl.gov/csep.html`.

7. Acknowledgments

The original mutational meltdown simulator was written in C by Michael Lynch. The first parallel simulator was developed in MPL by John Conery. The current simulator is the result of useful suggestions and hard work by David Butcher, Jeff Holmes, Tommy Hovland, and Sam Jones. The mutational meltdown project has been supported by grants from the Data-Parallel Research Initiative (DEC, MasPar, and Thinking Machines), the Murdock Charitable Trust, the Oregon Advanced Computing Institute, and the National Science Foundation (BSR 8911038 and BSR 9024977). We are also very grateful to Prof. Trond Steihaug and Jan Henriksen of Para//AB, the supercomputer center at the Institutt for Informatikk, University of Bergen, Norway, for their support and the generous use of their 16,384-processor MP-2216 for our longer simulation experiments.

References

[1] Biagioni, E. S. and Prins, J. F. Scan-directed load balancing for mesh-connected highly parallel computers. In *Unstructured Scientific Computation on Scalable Multiprocessors*. MIT Press, 1992.

[2] Blank, T. The MasPar MP-1 architecture. *COMPCON*, Feb. 1990.

[3] Conery, J. S. et al. A parallel algorithm for calculating the free energy in DNA. To appear in *Proceedings of the Hawaii International Conference on System Sciences* (Maui, Jan. 3–6), 1995.

[4] Giles, R. and Tamayo, P. A parallel scalable approach to short-range molecular dynamics on the CM-5. *Proc. Scalable High Performance Computing Conference*, 1992, pp. 240–245.

[5] Hatcher, P. J., Quinn, M. J., et al. Architecture-independent scientific programming in Dataparallel-C: Three case studies. *Supercomputing '91*, pp. 208–217.

[6] Karypis, G. and Kumar, V. Unstructured tree search on SIMD parallel computers: A summary of results. *Supercomputing '92*, pp. 453–462.

[7] Lynch, M., Conery, J.S., and Bürger, R. Mutational meltdowns in sexual populations. *Evolution* (accepted for publication).

[8] Myczkowski, J. and Steele, G. Seismic modeling at 14 gigaflops on the Connection Machine. *Supercomputing '91*, pp. 316–325.

[9] Nickolls, J. R. The design of the MasPar MP-1: A cost effective massively parallel computer. *COMPCON*, Feb. 1990, pp. 25–28.

[10] Press, W. H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[11] Stone, H. S. *High Performance Computer Architecture* (3d ed). Addison-Wesley, 1993.

[12] Zima, H. and Chapman, B. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.