Explicit SIMD Programming for Asynchronous Applications*

Andrea Di Blas   Richard Hughey
Department of Computer Engineering
Baskin School of Engineering
University of California, Santa Cruz
{andrea|rph}@cse.ucsc.edu

Abstract

This paper presents the SIMD Phase Programming Model*, a simple approach to solving asynchronous, irregular problems on massively parallel SIMD computers. The novelty of this model consists of a simple, clear method on how to turn a general serial program into an explicitly parallel one for a SIMD machine, transferring a portion of the flow control into the single PEs. Three case studies (the Mandelbrot Set, the N-Queen problem, and a Hopfield neural network that approximates the maximum clique in a graph) will be presented, implemented on two different SIMD computers (the UCSC Kestrel and the MasPar MP-2). Our results so far show good performance with respect to conventional serial CPU computing time and in terms of the high parallel speedup and efficiency achieved.*

1. Introduction

Since Flynn's classification of computer architectures [8], parallel computers have developed along two major structures: the Single-Instruction Multiple-Data (SIMD) and Multiple-Instruction Multiple-Data (MIMD). Parallel applications, on the other hand, can be divided into three main categories [24]:

- synchronous problems, with a uniform structure and an easy load balancing;
- loosely synchronous problems, in which a synchronization phase must be provided to assure proper computation flow and load distribution; and
- asynchronous problems, in which the convergence rate is data-dependent and therefore unpredictable.

Synchronous problems naturally map onto SIMD architectures and take full advantage of their regular structure. Traditionally however, it has been more common to develop algorithms for loosely synchronous and asynchronous problems on the more flexible MIMD architecture. In this paper we will describe a programming technique, called *SIMD Phase Programming Model* (SPPM), that provides a simple yet efficient framework for developing asynchronous, general programs on SIMD platforms. This work originates from our experimental studies with the Kestrel parallel processor, a SIMD computer designed and in use at the University of California, Santa Cruz. Our model, however, seems to be applicable to any SIMD computer featuring at least the following basic forms of processor autonomy [10, 21]: i) local indirect addressing, ii) PE masking capability, and iii) a global signal (e.g., a "global-or") for PE polling and conditional branching. It must be noted that this is a

*In *IEEE Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP2000)*, July 2000, pp. 258–267.

"work in progress"; the model is further refined with each new application. Here we will present experimental results from the Kestrel processor and the MasPar MP-2 on three asynchronous problems with different degrees of irregularity and data connectivity.

2. Related work

Apart from a large number of papers and books dealing with systolic processing, little can be found in the literature about solving irregular, general problems on SIMD architectures. The various approaches can be grouped into three categories: i) single case studies, ii) high-level support systems and parallelizing compilers, and iii) "MIMD on SIMD".

Many researchers have successfully explored the use of SIMD processors for solving irregular problems, like stochastic simulation [4], fractal image compression [14], image processing [20], graph matching [2] and general scientific computing [7]. The work by Tombulian and Pappas proposes a solution to the Mandelbrot Set on a MasPar MP-1 [25]. Tong and Leung offer examples of solutions of asynchronous problems requiring backtracking (like the N-Queen problem) implementing finite domain constraint languages on massively parallel SIMD machines [26]. All of these works, however, present a specific solution to a specific problem on a specific architecture, and therefore lack in generality.

A more general approach to deal with loosely synchronous and asynchronous problems can be found in works like the one by Shu and Wu [24]. Their view of a program as a collection of *processes* consisting of *atomic computations* closely matches our organization of a program in *cycles* and *phases*. Their computation selection algorithms and ours are also similar in principle. The major difference lies in their use of a high-level runtime support system (the P Kernel), as opposite to our low-level program coding methodology that does not need a high-level scheduler. In fact, our approach favors explicit versus implicit parallel programming, which may be slightly more difficult but usually achieves better performance, not relying on a compiler to parallelize the code [9]. In a subsequent paper [29], Wu and Shu give a comprehensive survey of the different approaches to general-purpose programming on SIMD architectures, and Üresin and Dubois [27] offer a mathematical analysis of asynchronous implementation of iterative algorithms. Again, these approaches are inevitably bound to the availability of the runtime support system and/or the parallelizing compiler on the architecture at hand.

Other approaches to the implementation of asynchronous algorithms on SIMD systems fall under the category of "MIMD on SIMD" [1, 28], and make use of the local indirect addressing capabilities to load the program along with the data in the PEs' local memory. Functional parallelism is then achieved by means of a local program counter, but this actually turns the SIMD system into a MIMD one. Finally, the general problem of mapping parallel algorithms to different parallel architectures, even though the main focus is on MIMD systems, is discussed thoroughly in the paper by Chaudhary and Aggarwal [3].

3. The "SIMD Phase Programming Model"

The computational model for SIMD Phase Programming can be viewed as an enhanced "instruction-level approach" [24, 29]. A parallel computation will be split into a set of *phases*, "atomic" sequences of instructions serially executed by the PEs. The phases are grouped at a higher level into one or more different *cycles*. The array controller loops through all the instructions within a cycle, one phase after the other. Each PE stores

in an appropriate register (the "cycle register") a number corresponding to the phase it is executing or going to enter next. At the beginning of each phase, a simple *activation check* defines what the *active set* of PEs will be for that phase. After the execution of the instructions in the phase body, a *transition test* will assign the next phase to each individual PE, based on their computation outcome. Therefore, self-synchronization of operation is performed within each PE by proper phase assignment in the cycle register, not requiring any upper-level supervision.

The basic SPPM parallelization process has four steps. The first step consists of finding a way to partition the problem (either the data or the computation) into a number of possibly disjoint subproblems (tasks), creating a "task pool". At the beginning of the computation, each PE will be assigned a task to perform, removing it from the pool. During the computation, the remaining tasks are assigned on demand to the PEs upon completion of their tasks. When the task pool is empty, the PEs cannot receive new tasks and simply sit idle, waiting for the others to finish. This wasted time is called "tailing overhead", and can be reduced by creating a number of tasks much larger than the number of available PEs (Section 6). The computation ends when the last PE terminates the last task.

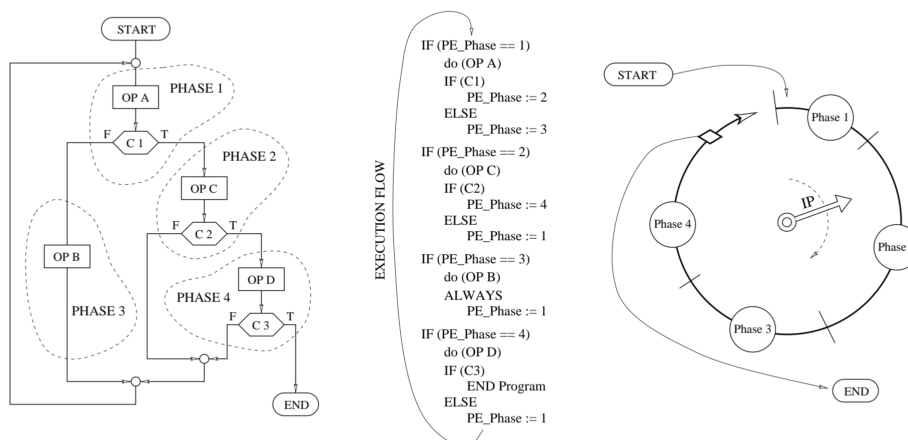

Figure 1. SPPM parallelization: serial flow-chart with phase identification (left), parallelized program (center), "phase-clock diagram" (right).

The second step in creating an SPPM program is to write a serial program reflecting the chosen problem partitioning (Figure 1, left). Then, the major branching conditions are identified, and used to partition the program into "phases", composed of operational blocks and their respective trailing branching conditions. Phases are assigned a unique number.

The third step is the actual parallelization, and consists of adding "activation checks" at the beginning of phases, before the operational instructions, and of turning the branching conditions into phase register assignments (Figure 1, center). At this point, the program has become a simple sequence of phases and is basically parallelized. Figure 1, right, shows the "phase-clock diagram" representation of the parallel program, symbolizing that the instruction pointer (IP) in the controller will loop through the program over and over, broadcasting the instructions of all phases in sequence, until a terminating condition is met. Of course, in this process, we introduce different sources of overhead, of which, by far, the major is the intrinsic parallelism overhead, originating from the fraction of PEs that are non-active in a phase.

Thus, the fourth step in building an SPPM program is reducing parallelism overhead,

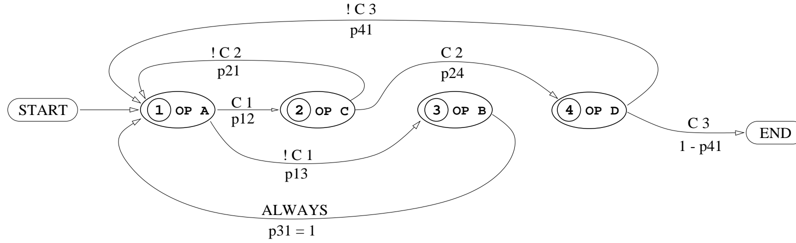

Figure 2. The dependency graph for the example in Fig. 1.

which means maximizing the number of logically active PEs over time. A first method to improve the performance is simply to order the phases in such a way that, according to the statistical behavior of the algorithm, the probability for a PE active in phase n to be active also in phase $n+1$ is maximized. A useful means of achieving this is to obtain a dependency graph (Figure 2) with transition probabilities.

Unfortunately, dealing with irregular problems, this is not enough, and some optimization techniques can be adopted to further speed up the execution time.

4. Optimization

Dynamic phase skipping At the beginning of a phase, before the activation check, we can poll the PEs to ensure that at least one will be active in that phase. Otherwise, the phase (or group of consecutive phases, if program structure allows) can be skipped. This kind of polling can be quickly performed via a `globalor` signal. Differently from other results [1, 12], we found that if phases are considered as macro-instructions, they are not always needed even though the number of PEs is large, and therefore this method actually improves performance.

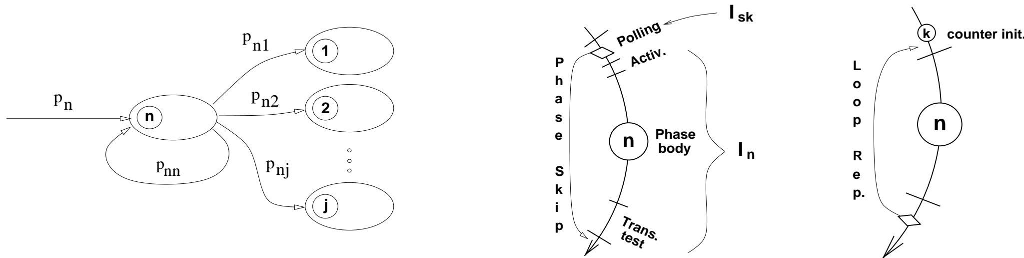

Figure 3. *Phase skipping* **and** *loop repeating* **optimizations.**

Deciding whether to apply phase skipping is based on the probability of a phase to be executed and on its execution cost. Referring to Figure 3, center, let I_{sk} be the cost of the polling and I_n be the cost of the phase itself. The total overhead resulting from applying phase skipping to a phase for l loops through the cycle is lI_{sk}. For simplicity, let p_n be the probability that at least one PE be active in the phase. Then the total benefit A from applying phase skipping is the difference between the instructions skipped and the overhead cost $A = l(1-p_n)I_n - lI_{sk}$. In fact, the polling instructions will be executed at every loop and, on the average, a $1-p_n$ fraction of the loops the phase instructions will be skipped. As I_{sk} depends on the machine architecture and the probability p_n depends on the problem and algorithm, the criteria to decide whether applying phase skipping to a phase becomes $I_n > I_{sk}/(1-p_n)$ A further improvement can be obtained by counting the number

of PEs that would be accessing a phase and entering the phase only if the number is above a certain threshold, which must be dynamically adjusted because toward the end of the computation the actual number of PEs still involved in the process decreases. However, this more complex polling may introduce a larger overhead, reducing its applicability.

Static loop repeating Looping through a phase or a group of consecutive phases a fixed number of times before moving on to subsequent phases can also improve performance. For this method to be useful, a significant number of PEs must be active in the same phase many times in a row, which is often the case with a program's main computation. Referring to Figure 3, this means that $p_{nn} \gg \sum_i p_{ni}$. The optimal number of repetitions can be figured out experimentally and tailored to the specific problem. However, when enough knowledge of the problem is available, one can loop through a phase while, on the average, at least half of the PEs are still active. Let Np_n (where N is the total number of PEs and p_n is the phase probability) be the average number of PEs that are going to be active in Phase n. After k loops through the phase, $Np_n\, p_{nn}^k$ are still active. The number k^* of loops after which the number of active PEs is down to $N/2$ is therefore $k^* = \lceil -\frac{\log 2p_n}{\log p_{nn}} \rceil$ As in the case of phase skipping, a dynamic adjustment can provide better performance, based on the current number of active PEs, but only when the controlled program section is large enough to justify the increased polling cost.

Multiple cycles Phase skipping and loop repeating are performed within a single cycle. SPPM programs can also be partitioned at a higher level, combining phases into different cycles. Moving between cycles is accomplished with "jump phases", in which a test for a cycle jump will be performed, based on the global-or or a more complex polling with dynamic threshold. After a cycle jump, the instruction pointer will loop through the new cycle over and over until another cycle jump brings it back to the previous cycle or moves it to yet another one. For example, multiple cycles are useful for large phases in which PEs are seldom active. Figure 4 shows an example of this, assuming that Phase 4 in Figure 1 is an output phase performed only at the end of a much heavier computation. The phase is removed from the main cycle and turned into a separate cycle, including the program terminating condition. Different phase registers must be used in different cycles to allow proper computation flow.

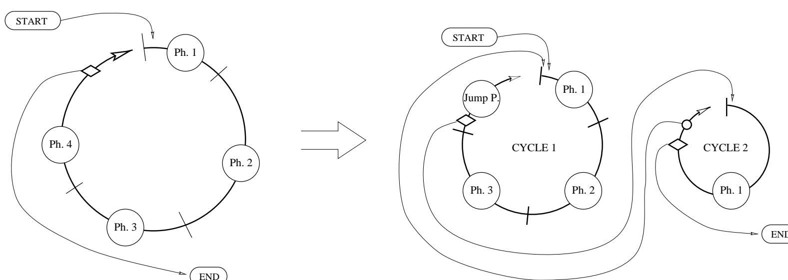

Figure 4. Splitting a cycle into multiple cycles.

The organization of a program into multiple cycles is not properly an optimization, but provides, however, a much cleaner program partitioning and structuring that is especially useful for large programs.

5. The UCSC Kestrel and the MasPar MP-2

The Kestrel parallel processor Kestrel (born for computational biology applications [15]) is a 512-PE SIMD linear array on a single PCI board for a Linux/NT/OSF host [5, 11]. The system (Figure 5, left) is composed of a PCI interface unit, an instruction memory, the array controller, and the PE array. Input and output of data to and from the array can take place only at the array ends, through the array controller, with input and output queues providing synchronization with the PCI bus. Each PE (Figure 5, right) is composed of 3 basic blocks: an operating unit, a local memory and two banks of "systolic shared registers" [16]. The 8-bit operating unit groups an ALU, a multiplier, a bit-shifter (also used for conditional operation and PE masking), a comparator and some flag logic. The local memory is a 256-byte static RAM (operating at the same speed as the ALU) that can be addressed using local indirection. A global *wired-or* signal (like the `globalor` in MasPar MPL) can be detected by the controller and is used to poll the PEs and perform global branching in the instruction sequence.

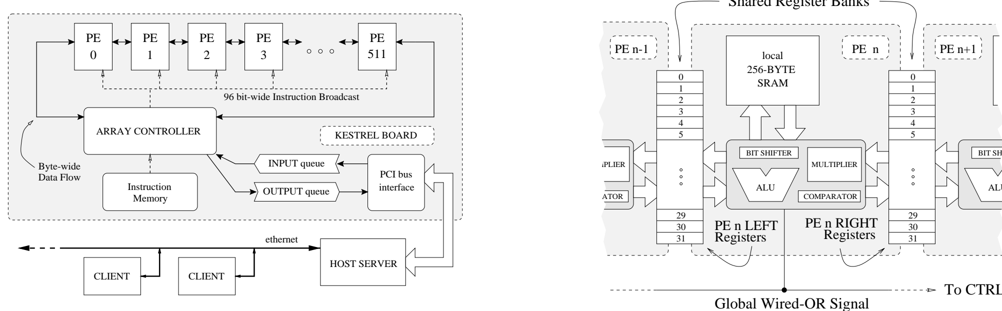

Figure 5. Kestrel high-level structure (left) and a detail of the PE array (right).

The MasPar MP-2 The MasPar is a well-known parallel processor. It is a SIMD bi-dimensional array composed of 1k, 2k, 4k, 8k or 16k PEs (our programs were tested using a 4k-PE MP-2). Its operating mode complies with our requirement on processor autonomy as stated in Section 1 and is therefore suitable to implement programs according to our model. The MasPar MP-2 architecture is essentially identical to the MasPar MP-1 architecture [22], except that the word length is 32 bits instead of four and the bandwidth to local memory is doubled.

6. Case studies

The Mandelbrot Set is on the complex plane, and is obtained by performing an iteration on each pixel and assigning it a color according to when the iteration stops [6]. This problem is a classic example of an irregular work load with "shared-nothing" data [9, 25].

The N-Queen Problem is to place N queens on an $N \times N$ chess board in such a way that no queen can be taken by any other, that is, no queen can be on the same row, column or either diagonal as any other. Our approach to the problem requires finding all possible solutions rather than just the first one, to effectively compare overall performance (in a depth-first search like this, super-linear speedup can be achieved when looking for just any one solution, due to the non-uniform distribution of solutions in the search space [23]). This problem is another example of an irregular, data-dependent workload [24, 26].

A *clique* in a graph is a completely connected subgraph. The maximum clique in a graph is the largest clique in the graph, and finding it is a problem known to be NP-hard. The third case study is a Hopfield neural network used to approximate the maximum clique. The algorithm uses multiple adaptive restarts to focus on the most promising areas for the network evolution [18, 19]. During network evolution, an energy function decreases to a minimum, corresponding to a clique. The lower the minimum, the larger the clique. This algorithm is significantly more difficult to parallelize, mainly because the data space cannot be partitioned into disjoint sub-spaces. The SPPM implementation, therefore, partitions the computation process itself, rather than the data.

Load balancing and task allocation In both the Mandelbrot Set and the N-Queen Problem, it is possible to partition the problem into a number of disjoint subproblems. The natural choice for these subproblems in the Mandelbrot Set are square $n \times n$ pixel "patches". In the N-Queen Problem, the tasks are the sub-trees corresponding to the valid initial positions of M queens in the first M columns of an $N \times N$ board. The Hopfield network problem must be treated differently. Without resorting to complex divide-and-conquer strategies, the graph (the network) cannot be partitioned into disjoint subgraphs. We therefore choose to partition the computation process. Instead of performing a certain number of restarts on a single processor, we perform a fraction of the restarts on multiple processors, cutting down execution time for the same total number of restarts. The random nature of the algorithm allows different PEs to explore different zones of the graph and come up with different solutions. The basic mechanism, however, is still the same as in the other case studies.

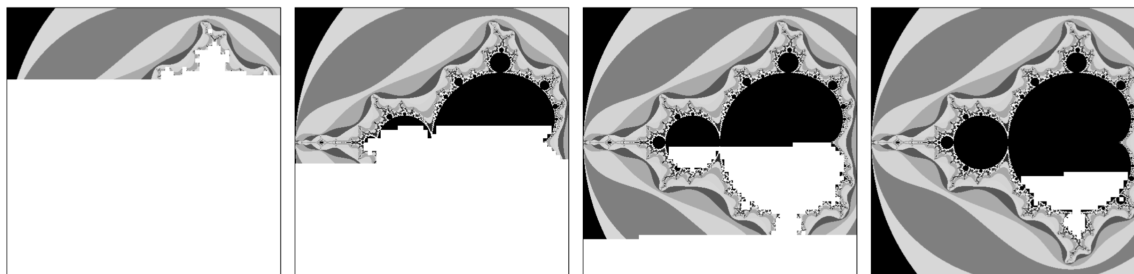

Figure 6. Mandelbrot Set computation at four time steps during program execution.

Load balancing is dynamically performed as long as the number of subproblems is larger than the total number of PEs. The Mandelbrot Set computation progress is a good example of this behavior (Figure 6). If the granularity (ratio between the number of subproblems and the number of PEs) is correctly chosen, all PEs will be logically active until the very end of the computation, keeping the "tailing-overhead" low. A steeper edge (corresponding to a better utilization) can be obtained by increasing the granularity. However, beyond a certain limit, the benefits are overcome by the increase in other sources of overhead (especially the communication and I/O overhead). Figure 7, left, shows the utilization curves obtained with the 12-Queen program for different granularities. It can be clearly seen how a low granularity (curve M = 3) poorly exploits the parallelism, and how, on the other hand, increasing the granularity too much (curve M = 5) brings other sources of overhead into play and degrades performance.

Results Table 1 shows the wall clock execution times of the three case studies with different parameters and problem sizes. In the case of the maximum clique, the program was written only for the MasPar system because the limited local memory on Kestrel would

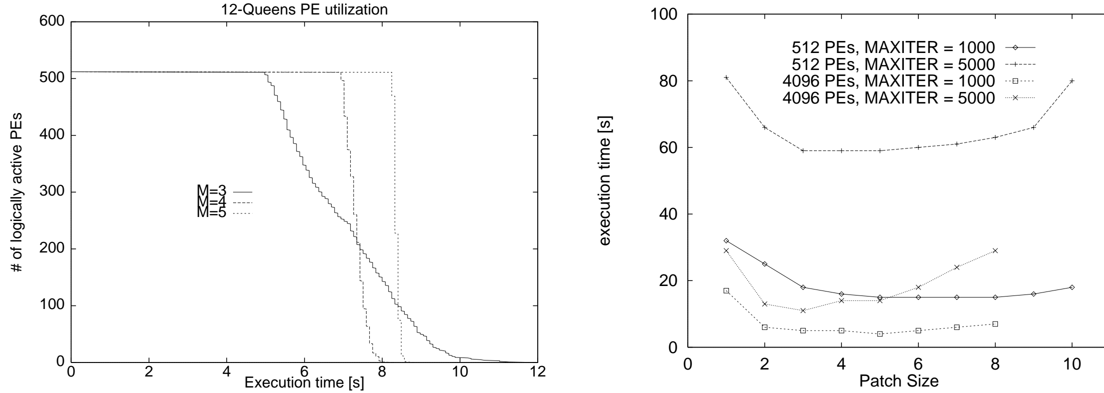

Figure 7. 12-Queen problem PE utilization with different granularities (left) and Mandelbrot Set wall-clock timings versus problem granularity (right).

have allowed only the processing of graphs too small to be of interest. The example graphs are from the official benchmarks of the DIMACS contest [13, 17]. For comparison, the timings for a program running on a serial CPU (143MHz SUN Ultra-SPARC 1 with 256 MB RAM) are also reported, along with the performance speedup of the parallel programs.

MAXITER	CPU time [s]	Kestrel (512 PEs) time [s]	Kestrel (512 PEs) speedup	MP-2 (4096 PEs) time [s]	MP-2 (4096 PEs) speedup
1000	22.6	1.8	12.0	5	4.5
5000	104.9	6.2	16.9	13	8.0
50000	1052.2	52.9	19.9	88	11.9

(a) Mandelbrot Set (with MAXITER iterations inside the set).

N	solutions found	CPU time [s]	initial branches	Kestrel (512 PEs) time [s]	Kestrel (512 PEs) s.u.	MP-2 (4096 PEs) time [s]	MP-2 (4096 PEs) s.u.
13	73712	35.5	6404	4.1	8.6	16	2.2
14	365596	226.0	9632	23.4	9.6	80	2.8
15	2279184	1596.3	13980	156.3	10.2	495	3.2

(b) N-Queens.

graph	$\|V\|$	$\|E\|$	γ^*	best γ found	CPU time [s]	MP-2 time [s]	s.u
san200_0.9_1	200	17910	70	70	290.2	12	24.1
hamming8-4	256	20864	16	16	13.1	1	13.0
p_hat500-3	500	93800	?	44	354.3	8	44.2

(c) Maximum Clique with Hopfield network ($|V|$, $|E|$ and γ^* are the graph size, the number of edges and the size of the maximum clique respectively).

Table 1. Execution time of case study programs and speedups with respect to CPU

As expected, the parallel programs perform best in the Mandelbrot case, which has the least communication and synchronization. In all three problems, performance with respect to a serial processor increases with increasing problem complexity because the parallelism is increased and parallelization overhead is reduced.

Parallel speedup and efficiency. The parallel speedup is the ratio between the

execution time using a single PE and the execution time using N PEs, which gives a measure of how *scalable* the program is with respect to machine size. The efficiency is the ratio between the speedup and the corresponding number N of PEs used, and indicates the average fraction of PE active time with respect to total time (Figure 8). As mentioned above, the curves relative to the maximum clique program exhibit super-linear speedup at times, and therefore the corresponding efficiency is greater than one. The linear portion of the speedup graphs, before saturation, extends well into high numbers of PEs, indicating that the programming model provided good scalability.

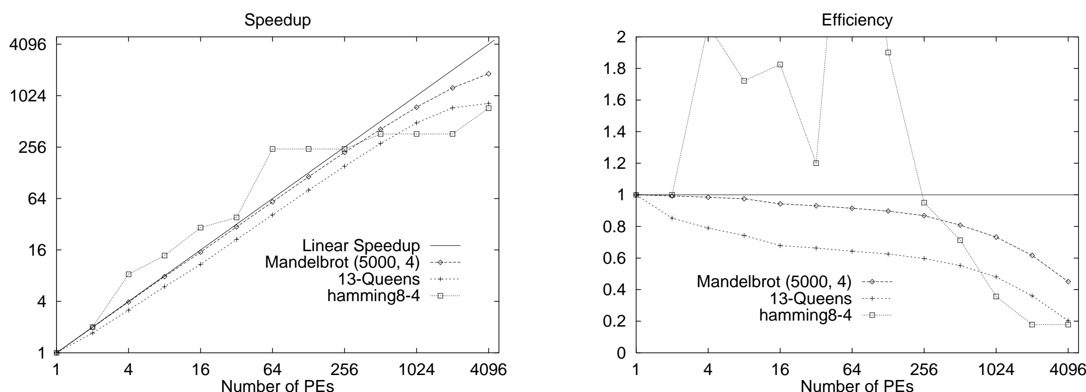

Figure 8. Speedup (left) and efficiency (right) of the case studies.

7. Discussion and conclusions

Even from this informal description of the proposed programming model, two major drawbacks stand out. First, finding a good problem partitioning may not always be easy or even possible. Moreover, if the granularity cannot be varied, the amount of local memory within the PEs can be a limit to the problem size. A solution to this can be "PE clustering", grouping more than one PE into a single operational unit. Second, the parameters for the optimization techniques described above strongly depend on the statistical behavior of the algorithm on the data, and thus require a certain amount of experimental trimming. The development of profiling tools could ease this task. It must be noted, however, that these two problems are common among explicit programming methodologies, and that they were easily overcome in our case studies. Currently, we do not foresee SPPM to be a compiler method. Instead, it is a framework for describing and implementing explicitly parallel programs quickly. Its use enables us to envision a variety of coding and analysis tools to assist the programmer.

To summarize, the main result of this work is a further proof that SIMD parallel computers are not so restricted in their applicability, but can also efficiently solve asynchronous, general problems. A simple programming paradigm, the SIMD Phase Programming Model, has proved so far to be an adequate tool to perform this task.

Acknowledgments The authors are thankful to Arun Jagota for his Hopfield neural network algorithm and his assistance in the development of its parallel implementation, and to Kevin Karplus and Leslie Grate for many fruitful discussions. This work was supported in part by the NFS grants MIP-9423985, EIA-9722730, and CDA-9115268.

References

[1] N. B. Abu-Ghazaleh, P. A. Wilsey, X. Fan, and D. A. Hensgen, "Synthesizing variable instruction issue interpreters for implementing functional parallelism on SIMD computers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, pp. 412–423, Apr. 1997.

[2] R. Allen, L. Cinque, S. Tanimoto, L. Shapiro, and D. Yasuda, "A parallel algorithm for graph matching and its MasPar implementation," *IEEE Trans. on Par. and Dist. Sys.*, vol. 8, pp. 490–501, May 1997.

[3] V. Chaudhary and J. K. Aggarwal, "A generalized scheme for mapping parallel algorithms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 328–346, Mar. 1993.

[4] J. S. Conery, M. Lynch, and T. Hovland, "Optimizing irregular computations on SIMD machines: A case study," in *The Fifth Symp. on the Frontiers of Massively Parallel Comp.*, pp. 222–230, Feb. 1995.

[5] D. Dahle, J. D. Hirschberg, K. Karplus, H. Keller, E. Rice, D. Speck, D. H. Williams, and R. Hughey, "Kestrel: Design of an 8-bit SIMD parallel processor," in *Seventeenth Conf. on Advanced Research in VLSI*, pp. 145–162, Sept. 1997.

[6] A. K. Dewdney, "A computer microscope zooms in for a look at the most complex object in mathematics," *Scientific American*, pp. 16–21, Aug. 1985.

[7] J. R. Fischer, L. E. Hamet, C. M. Mobarry, J. A. Pedelty, J. S. Cohen, R. K. de Fainchtein, B. A. Fryxell, P. J. MacNeice, K. M. Olson, and T. L. Sterling, "The practicality of SIMD for scientific computing," in *The Fifth Symp. on the Frontiers of Massively Parallel Comp.*, pp. 258–264, Feb. 1995.

[8] M. J. Flynn, "Very high speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.

[9] V. W. Freeh, "A comparison of implicit and explicit parallel programming," *J. of Parallel and Distributed Computing*, vol. 34, pp. 50–65, 1996.

[10] D. M. Hawver and G. B. Adams III, "Processor autonomy and its effect on parallel program execution," in *The Sixth Symp. on the Frontiers of Massively Parallel Computation*, pp. 144–153, Oct. 1996.

[11] J. D. Hirschberg, D. M. Dahle, K. Karplus, D. Speck, and R. Hughey, "Kestrel: A programmable array for sequence analysis," *J. of VLSI Signal Processing*, vol. 19, pp. 115–126, 1998.

[12] D. Hollinden, D. Hensgen, and P. A. Wilsey, "Experiences implementing the MINTABS system on a MasPar MP-1," in *The Sixth Symp. on the Experiences with Distributed and Multiprocessor Systems*, pp. 43–58, Mar. 1992.

[13] Home page of *Center for Discrete Mathematics and Theoretical Computer Science.* `http://dimacs.rutgers.edu`.

[14] C. Hufnagl, J. Hämmerle, A. Pommer, A. Uhl, and M. Vajteršic, "Fractal image compression on massively parallel arrays," in *PCS'97: The 1997 Picture Coding Symp.*, pp. 77–80, Sept. 1997.

[15] R. Hughey, "Parallel hardware for sequence comparison and alignment," *CABIOS*, vol. 12, no. 6, pp. 473–479, 1996.

[16] R. Hughey and D. P. Lopresti, "B-SYS: A 470-processor programmable systolic array," in *Int. Conf. on Parallel Processing*, vol. 1, pp. 580–583, Aug. 1991.

[17] A. Jagota, L. Sanchis, and R. Ganesan, "Approximately solving maximum clique using neural network and related heuristics," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26, pp. 169–204, American Mathematical Society, 1996.

[18] A. Jagota, "Approximating maximum clique with a Hopfield network," *IEEE Trans. on Neural Networks*, vol. 6, pp. 724–735, May 1995.

[19] A. Jagota, "An adaptive, multiple restarts neural network algorithm for graph coloring," *European J. of Operational Research*, vol. 93, pp. 257–270, 1996.

[20] S. Kyo, S. Okazaki, Y. Fujita, and N. Yamashita, "A parallelizing method for implementing image processing tasks on SIMD linear processor arrays," in *Computer Architectures for Machine Perception 1997*, pp. 180–184, Oct. 1997.

[21] P. J. Narayanan, "Processor autonomy on SIMD architectures," in *ICS'93: The Int. Conf. on Supercomputing*, pp. 127–136, July 1993.

[22] J. R. Nickolls, "The design of the Maspar MP-1: A cost effective massively parallel computer," in *COMPCON Spring 1990*, pp. 25–28, IEEE Computer Society Press, Feb. 1990.

[23] V. N. Rao and V. Kumar, "On the efficiency of parallel backtracking," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 427–437, Apr. 1993.

[24] W. Shu and M. Wu, "Asynchronous problems on SIMD parallel computers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, pp. 704–713, July 1995.

[25] S. Tombulian and M. Pappas, "Indirect addressing and load balancing for faster solution to mandelbrot set on SIMD architectures," in *The Third Symp. on the Frontiers of Massively Parallel Computation*, pp. 443–450, Oct. 1990.

[26] B. Tong and H. Leung, "Performance of a data-parallel concurrent constraint programming system," *J. of Logic Programming*, May 1998.

[27] A. Üresin and M. Dubois, "Parallel asynchronous algorithms for discrete data," *J. of the Association for Computing Machinery*, vol. 37, pp. 588–606, July 1990.

[28] P. A. Wilsey, D. A. Hensgen, N. B. Abu-Ghazaleh, C. E. Slusher, and D. Y. Hollinden, "The concurrent execution of non-communicating programs on SIMD processors," in *The Fourth Symp. on the Frontiers of Massively Parallel Computation*, pp. 443–450, Oct. 1992.

[29] M.-Y. Wu and W. Shu, "MIMD programs on SIMD architectures," in *The Sixth Symp. on the Frontiers of Massively Parallel Computation*, pp. 162–170, Oct. 1996.