Software Support for Non-Numerical Computing on Multi-core Chips

Jerry Potter
Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, CO 80523-1373 USA

Howard Jay Siegel
Department of Electrical and Computer Engineering
and Department of Computer Science
Colorado State University
Fort Collins, CO 80523-1373 USA

Abstract - *Multi-core chips present a new computing environment that can benefit from software support for non-numerical applications. Heterogeneous cores will allow efficient sophisticated multi-level parallel processing. Techniques are described that enable the association of the elements of related heterogeneous SIMD vectors. These techniques can be used for processing arrays of records, dynamic allocation of a core's memory and for imposing a multitasking layer of parallelism over a data parallel layer. Some of the operating system and hardware modifications needed to support these techniques are discussed. A version of the Smith Waterman algorithm for DNA sequence comparison was investigated briefly to study some of the advantages of an associative bit-serial core.*

Keywords: associative processing, multi-core chips, multi-level parallelism, parallel processing, SIMD

1 Introduction

Multi-core chips have been demonstrated to be very effective for data intensive applications such as graphics. An important question is: "Can this computing power be extended to other areas?" Existing chips have only a few cores, but chips with 10s of cores have been produced and chips with 1000s of cores are envisioned. These "massive" multi-core architectures with on-chip caches are similar in nature to the Processor-In-Memory (PIM) [7] and Intelligent RAM (IRAM) [9] designs of the 1990s, and the bit serial SIMDs (single instruction, multiple data stream [3]) of the 1970/80s such as the Massively Parallel Processor (MPP) [12] and Connection Machine (CM) [5]. The success or failure of a new architecture is often decided by the software model upon which it is based. Specifically, the PIM, IRAM, MPP and CM architectures were hampered because there was no appropriate software support for fine grain or data parallelism.

Heterogeneous multi-core chips with on-chip memory provide an opportunity to address the problems of large-scale multi-level parallel processing. In addition to having multiple cores, each core can exploit vector SIMD operations.

This paper describes how the associative computing model can provide data parallel [14] as well as multitasking software support for pattern matching and other non-numerical problems. Languages based on the associative model such as ASC [11] are easy to use and support multilevel parallelism, but require some operating system enhancements and minor hardware improvements to future multi-core designs. These modifications are discussed in this paper.

2 Background

Many non-numerical applications, including bioinformatics, database and web searching, rely more on pattern matching than on processing homogeneous arrays of vectors as is typical of graphics processing. These applications also process large volumes of data and can benefit from multi-core SIMD chips, but graphic processor instruction sets are designed for processing arrays of vectors of homogeneous data of standard numerical data types – i.e., bytes, integers, floats, etc. In contrast, most pattern matching algorithms require processing heterogeneous vectors of different precisions and types.

Many languages, such as C++, organize related heterogeneous data into structures or records that in turn are collected into files. Searching data organized as records with vector hardware can result in "baroque" expressions[1], due in part to the fact that structures organize data orthogonal to the desired direction for vector processing (as shown in Figure 1). Each vector consists of homogeneous data (data of the same type),

This research was supported in part by the Colorado State University George T. Abell Endowment.

[1] The IBM Cell Broadband Engine Programming Handbook spends almost 50 pages on how to develop SIMD programs (p. 619- 667).

where each vector in Figure 1 is a vertical collection of squares. Each record is a collection of (possibly) heterogeneous data fields (data items of different types), where each record in Figure 1 is a horizontal collection of squares. Thus, each vector contains items of the same type, one from each of the different records. Each record consists of items of (possibly) different types, one from each of the different vectors. SIMD programming of pattern matching and other non-numerical applications could be simplified by an approach that emphasized heterogeneous associations of homogeneous vectors instead of data structures consisting of homogeneous records of heterogeneous fields.

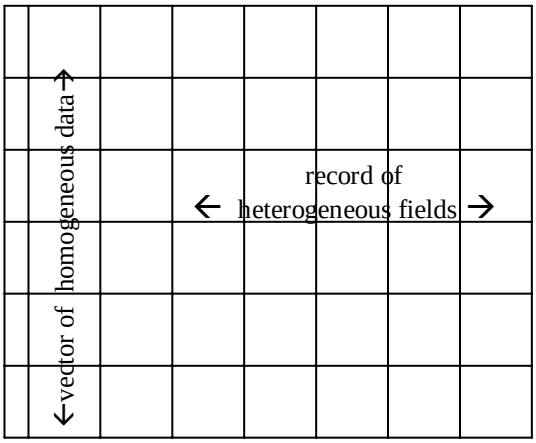

Figure 1 – Record versus Vector Organization

Moreover, some conventional operating system functions, such as memory allocation, were designed for records and are not efficient for processing large vectors across multiple cores. One possible solution is an Associative Data Parallel Operating System (ADPOS) for the chip's cores augmenting the existing OS on the control processor. This would support an OS/language family extension of the Linux/C++ family of software for SIMD cores that would improve performance of multi-core chips on non-numerical applications.

3 Associative computation

3.1 Associative cells

The associative model of computing is a data parallel model that assumes that there is a dedicated cell of processor and memory for every record of data even though the data is organized as vectors (as shown in Figure 1). In conventional languages, the size of the vectors must be maintained by the user and must be specified whenever a data parallel loop is to be executed. In associative computing, the operating system maintains the size of the vectors based on the number of records in memory at run time, i.e., the number of records is the size (length) of the data vectors (see Section 4.1). Because the size of the vectors is unknown at compile time, the programmer uses a $ notation to indicate that the entire vector is to be processed. Thus

```
b[$] = c[$] + d[$] ;
```

replaces

```
for (i=begin ; i<end ; i++)
  b[i] = c[i] + d[i] ;
```

Message Passing-Shared Memory Processor (MP-SMP) supercomputers can be difficult to program, as indicated in the following quote from the National Partnership for Advanced Computational Infrastructure (NPACI) Archives: "The teraflops system will be a Message Passing-Shared Memory Processor (MP-SMP) machine … An MP-SMP machine attempts to addresses (sic) both of these limitations, but this is accompanied by *an increase in programming complexity* (emphasis added)" [15].

As hardware becomes cheaper, the cost of programming becomes more significant. If the increase in chip capacity continues, shortly there will be thousands of processors on a chip and cost effective programming will be a major issue. Note that the NPACI article is concerned about the difficulty of programming 10s and 100s of processors not 1,000s and 10,000s. George A. Miller has observed [8] that people are limited in their ability to handle more than about seven tasks simultaneously.

One component of the increased complexity of programming supercomputers is that using conventional multitasking languages often requires breaking large loops into many smaller loops on different processors, greatly increasing the number of tasks that have to be coordinated. However, the associative model processes a vector in one logical step; no matter how large it is, using virtual cells when necessary. Thus, a file with 10,000 records maps directly onto 10,000 virtual cells.

3.2 Associative flag vectors

Because the index variable has been removed from the notation, associative computing uses flag vectors as an economical method of relating a field of a record in one vector with the associated field in a related vector. Consider a "payroll" data base of heterogeneous data as shown in Figure 2. The names are alphanumeric, social security numbers are integer and the salaries are floating point. Data records, such as these in data structure format, cannot, in general, be efficiently processed on vector SIMDs because 1) a homogeneous operation can not be applied to the various heterogeneous fields in the record, 2) records tend to be rather short for vector processing – several hundred fields at most, and 3) the

calculations on one vector cannot easily be associated with the other vectors.

Name	Social Security	Salary		Respond-ers
Smith	1234567	100,000.00		False
Jones	0987654	5,000.00		True
Potter	1112233	50,000.00		False

Figure 2 – An Associative Responders Flag Vector

However, given that you have a sufficiently large file of records, say 10,000, with the same format, they can be organized into vectors of 10,000 names, 10,000 social security numbers and 10,000 salaries. These vectors can be processed efficiently using vector operations, but a method is needed to allow the association of one element of a vector to be related to the corresponding element of a record in another vector.

Figure 2 shows an associative flag vector labeled Responders that is used to associate a record element in one vector with the corresponding record element in another vector.An associative flag vector replaces the need for indices for data parallel operations. For example, the C++ computation:

```
for (i=begin ; i<end ; i++)
  if  (Salary[i] >= 50000)
     Tax[i] = Salary[i] * 0.15 ;
  else
     Tax[i] = Salary[i] * 0.05 ;
```

would be replaced by

```
  if  (Salary[$] >= 50000)
     Tax[$] = Salary[$] * 0.15 ;
  else
     Tax[$] = Salary[$] * 0.05 ;
```

where the Responders flag vector keeps track of the comparison so that the True portion is executed for the True inequality results and the False portion is executed for the False inequality results.

Flag vectors can be used to indicate parallel vector to scalar reduction and the insertion of scalar data into a vector. If the Falkoff minimum field algorithm [1] is used to find the smallest salary, the result is an associative flag vector in the Responders field. The associative flag vector can be used in a reduction operation to extract a datum flagged by the vector. Thus, the reduction operation allows the logical association between the heterogeneous Name and Salary vectors to return the name associated with the salary found, i.e., "Jones," not the salary itself. This can be written as:

```
Name[minimum(Salary)] .
```

Similarly, related selected items can be changed by

```
Name[minimum(Salary)] = "Johnson".
```

Flag vectors are efficient and easy to use because one fixed size vector can replace a variable number of indices. If an array of indices is use to record a variable number of matches at random locations in a conventional architecture, scatter-gather hardware [6] (p. 73) is needed to perform vector operations.

3.3 Associative searching

Another reduction in the complexity of associative computer algorithms is due to parallel associative searching in place of indexing, pointing and linked lists. Data does not need to be sorted if efficient data parallel searching is available. More specifically, in a conventional computer, data records can only be organized by one key at a time: alphabetically, numerically, by date, etc., and the programmer must keep track of the ordering so that the data can be reordered when necessary. But when using data parallel associative searching, all fields can be efficiently searched without reordering the data. That is, the data in Figure 2 is unorganized, but the efficiency of the search for the minimum salary is unaffected.

The parallel searching capability allows new algorithms for difficult problems to be developed. For example, by definition, the processing of raw data requires searching of unorganized data. During the analysis of the data, multiple alternative organizational keys, or structure codes [11], can be added to the records without the need to sort or resort the data. Figure 3 illustrates a key for organizing the data based on the Name vector.

Name	Social Security	Salary	Key	Respond-ers
Smith	1234567	100,000.00	3	False
Jones	0987654	5,000.00	1	True
Potter	1112233	50,000.00	2	False

Figure 3 – An Organizational Key

3.4 Associative communication

The virtual associative cells are treated as separate processors in the associative model. Nearest neighbor communication between (virtual) processors is achieved by moving all items up or down equal amounts in unison. This is denoted by modifying the $ notation by an offset such as b[$+1] or c[$-5] assuming the data is sorted, either naturally when input, or by software after an organizational key has been established. This communication can be accomplished by a regular "shift" or data alignment operation. More complex communication patterns can be easily expressed by using

variables with this notation, e.g., b[$+offset]. The variable may be a scalar or a vector. If scalar, the same offset applies to all elements. If a vector, a different offset may be applied to each element. The hardware for the vector version may become prohibitively expensive for on-chip communication as this notation is equivalent to message passing.

4 Associative data parallel operating system

The trend toward on-chip memories addresses the long memory latency problem. But in order to use on-chip memory efficiently an ADPOS is needed. Conventional sequential and parallel operating systems are "pointer based" and accomplish memory management using linked lists, stacks and heaps. The single instruction stream aspect of associative parallelism does not work efficiently with pointer based systems. Consequently, an associative parallel approach to on-chip memory management that is compatible with the conventional operating systems of the chip's controller is an essential component of a comprehensive Parallel and Distributed Operating System (PDOS).

The PDOS must divide the on-chip memory management problem into two separate tasks: one for programs and one for parallel data. The program OS that must run on the chip's controller can be equivalent to a conventional Linux OS with the standard stack of activation records. The memory heap, however, would be for local scalars and constants only. The associative component would handle the parallel data and would 1) allocate the (virtual) memory-processor cells, 2) administer the associative multitasking component and 3) control data flow into and out of the cores' on-chip memories.

4.1 Memory Management

An associative Busy/Idle (B/I) flag vector is used to allocate and release cells. When a new record is input, the ADPOS allocates a new cell by setting the associated B/I vector bit to busy. When the cell is no longer needed it is released for reuse by setting the bit to idle, thus maintaining effective memory utilization without garbage collection. The B/I vector identifies the active cells that are to be processed by the $ notation in the SIMD expressions mentioned earlier.

When applied to multi-core SIMD chips, the associative B/I vector can be easily extended to virtual cells. In order to implement virtual cells, the OS need only to buffer two additional associative vectors (responders and results) with each page of data records. The responders vector maintains the logical status of the virtual cells, the results vector contains the temporary results and the B/I vector maintains memory allocation status.

As discussed in Section 3.1, because the ADPOS maintains a record of active cells, the inner most "for loop" is replaced by the $ notation, eliminating the need for the programmer to maintain a count of the number of records to be processed. This is in contrast to the MP-SMP model mentioned earlier where not only must the number of records be maintained but the parceling of those records to the various processors must be maintained, the communication between processors must be maintained, etc.

The associative B/I vector functionality can be expanded to support a hierarchical multitasking data parallel model of computing discussed in Section 4.2.

4.2 Multitasking

In the traditional associative SIMD model, where every record has a dedicated processor, some processors may be idle while others are busy. Multiple Associative Computing (MASC) addresses this issue [10]. As a simple example of its execution, assume a population can be divided into two mutually exclusive sets. For example, those citizens who earn $50K or more annually and those who earn less. Assume also that the tax rate for those earning less than $50K is 5% while the tax rate for those earning $50K or more is 15%. The traditional associative data parallel method for calculating the taxes of all citizens, as described in Section 3.2, is to select all citizens who earn less than $50K and calculate their taxes (the processors associated with those citizens that earn $50K or more are idle during this step) and then select all citizens who earn $50K or more and calculate their taxes (the processors associated with those citizens that earn less than $50K are idle during this step). That is, the program for both cases must be broadcast to all processors but is executed only by those which require it. As a result the processors are, on average, idle half of the time.

Because the citizens form mutually exclusive groups, both computations can take place simultaneously if hardware is provide to allow both instruction streams to be sent in parallel and each processor executes the instruction stream appropriate for it. Figure 4 illustrates how the Responders register can be augmented by a Task ID register to address this issue. ADPOS administers the processors by updating the Task ID register of a cell when a new task is assigned. The instruction stream of Figure 4 would carry the task IDs as well as the instructions. An instruction would only be executed by those cells with 1) the correct task ID and 2) a True responders bit. Expanding this technique to more than two cases reduces the problem with case statements, which Hennessy and Patterson [4] (p. 650) identify as a major drawback to SIMD computation.

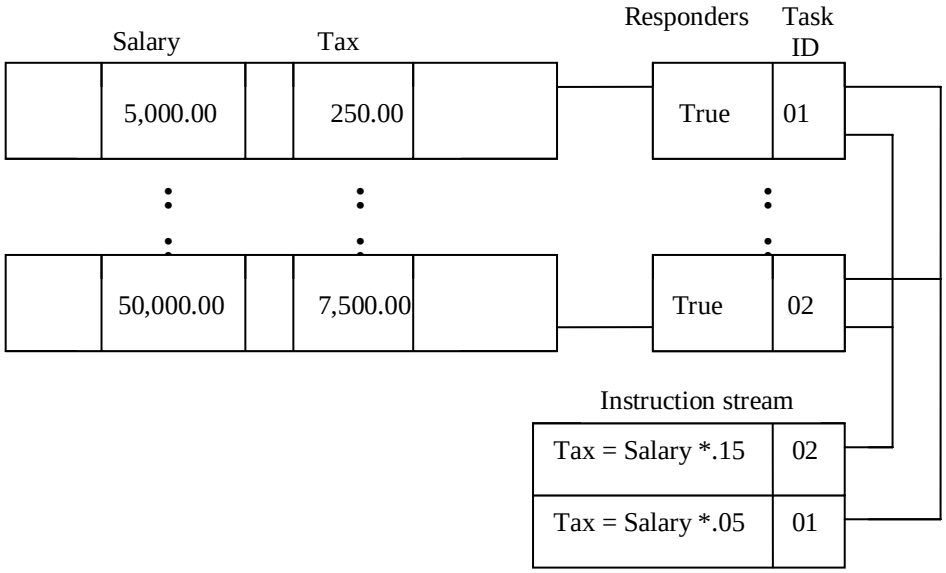

Figure 4 – MASC – A Multilevel Parallel Architecture

5 Hardware

Some hardware enhancements are needed to support full associative computing on multi-core chips, such as a complete on-chip controller capable of running a traditional multitasking OS so that the chip is a complete parallel processor, not an adjunct array processor. In more advanced applications, the "host" processor would become the I/O server for the multi-core chip. The need for on-chip communication was discussed briefly in Section 3.4. A brief discussion of hardware support for some of the other items follows.

5.1 Heterogeneous cores

Non-numerical applications would benefit from heterogeneous cores. That is, cores which are specialized for the different data types, such as an integer core, a 32 bit floating point core, etc. A bit serial core would be beneficial for the associative computing model. In particular, the Falkoff maximum value and related algorithms are used extensively and they work best with bit serial data. Conventional vector register cores can be used to find minimum and maximum values, but they can not find the least upper bound and greatest lower bound relative to a specified value in one pass as can the modified Falkoff algorithm [11] (p. 50). These operations are helpful when processing structure codes and organizational keys.

5.2 Flag vectors

Associative flag vectors allow the elements of one vector to be associated with the corresponding elements of another. Many multi-core designs utilize fixed size vector registers. Assuming a 128 bit vector register, a search on sixteen 8 bit data elements will perform 16 comparisons in parallel where each comparison is separated on 8 bit boundaries. While a search on eight 16 bit data elements will perform 8 comparisons in parallel. Because the fixed size vector register supports varying size fields, flag vector alignment between vectors of different types can be difficult. That is, a search on sixteen 8 bit data elements will yield a 16 bit flag vector. In order to maintain the element by element association with, say 32 bit floating point data, in other 128 bit registers, the hardware must separate the 16 bit flag vector into four 4 bit flags, one for each of the four 128 bit registers containing the associated 32 bit data. Special flag vector operations that can divide and allocate the bit flags appropriately are necessary.

5.3 Parallel to scalar reduction

The reduction operation, described in Section 3.2, is used frequently and needs to be supported in the hardware. A flag vector has a one (True) in the i-th position and zeros (False) elsewhere to signify to the hardware that the i-th element of the associated vector is to be extracted. More than one element of a vector can be flagged and with hardware support, the values of the associated vector can be extracted for processing one by one in a loop by "updating" the flag vector after every extraction.

```
VERTICAL_EXT[$]    = VERTICAL[$-1][J] – SIGMA;
VERTICAL_INT[$]    = VERTICAL[$-1][J] – RHOSIGMA;
VERTICAL_EXT[$]    > VERTICAL_INT[$]    ?
    VERTICAL[$,J]  = VERTICAL_EXT[$]    : VERTICAL[$,J] = VERTICAL_INT[$];
VERTICAL[$,J]      >  0                 ?
    VERTICAL[$,J]  = VERTICAL[$,J]      : VERTICAL[$,J] = 0;
HORIZONTAL_EXT[$]  = HORIZONTAL[$-1][J] – SIGMA;
HORIZONTAL_INT[$]  = HORIZONTAL[$-1][J] – RHOSIGMA;
HORIZONTAL_EXT[$]  > HORIZONTAL_INT[$]  ?
   HORIZONTAL[$,J] = HORIZONTAL_EXT[$]  : HORIZONTAL[$,J] = HORIZONTAL_INT[$];
HORIZONTAL[$,J]    >  0                 ?
   HORIZONTAL[$,J] = HORIZONTAL[$,J]    : HORIZONTAL[$,J] = 0;
DIAGONAL[$,J]      = DIAGONAL[$-1][J-1] + DELTA[SEQb[J]];
DIAGONAL[$,J]      > VERTICAL[$,J]      ?
    DIAGONAL[$,J]  = DIAGONAL[$,J]      : DIAGONAL[$,J] = VERTICAL[$,J];
DIAGONAL[$,J]      > HORIZONTAL[$,J]    ?
    DIAGONAL[$,J]  = DIAGONAL[$,J]      : DIAGONAL[$,J] = HORIZONTAL[$,J];
```

Figure 5 – Smith-Waterman Bit-Serial Data Parallel Pseudo Code

5.4 Branching operations

Additional associative instructions such as an efficient "branch on register zero" instruction are important. Considerable time can be saved if there is the capability to determine that all responders to a query are false and the associated code can be skipped. Some algorithms branch frequently over short distances so that a branch mechanism that does not disrupt the instruction pipeline would be most helpful. One possibility is a "branch" that simply inhibits the execution of the next 1 or 2 instructions rather than disrupting the pipeline. The maximum number of instructions to skip depends on the costs involved, such as the length of the pipe, etc.

5.5 Multitasking instruction stream

A multitasking instruction stream can be implemented in several different ways. For example, it could be implemented either physically with a "very wide instruction stream" or in the time domain where the instruction delivery time is much shorter than its execution time. ADPOS needs one or two instruction streams dedicated to administering the cores. Special control instructions for ADPOS that cannot be ignored by the cells would also be needed. For example, "listen to instruction stream n," start and halt.

Section 4.2 illustrates how an IF statement can be mapped onto multiple processors automatically by ADPOS. The associative computing model assumes that Miller's observation is correct and that a programmer can manage approximately seven tasks simultaneously. Thus a minimum of three instruction steams and a maximum of nine or ten would be reasonable.

6 Smith-Waterman algorithm

As a test of the effectiveness of an associative bit serial core, the Smith-Waterman algorithm for detecting the similar regions in two protein sequences was programmed in ASC, an associative computing emulation language described in [11]. The ASC program was used to verify the pseudo code design shown in Figure 5. The pseudo code was then used to estimate the execution time of the algorithm given the proposed associative model enhancements. One advantage of this implementation of the algorithm is that it minimizes branching. The code shown does not require any branching because it is based on a bit serial "select bits" operation represented by the C++ conditional operator [6] (p. 636). Accordingly, the multitasking aspect described in Section 4.2 was not needed and was not investigated.

It was estimated that with this design, using the associative instructions described in this paper, two sequences of 128 nucleotides each can be processed in approximately 14,336 instructions. On an 8-core chip at 4 Ghz this equates to about 36 billion cell updates per second. When adjusted for the number of cores and their speed, this is about 50% faster than the 3 billion cell updates per second reported in Farrar [2] [2].

In the optimized design, the overall execution time is dominated by the movement of data to and from the registers so that the advantage of a bit-serial multi-core implementation would be the ability to 1) avoid branching and the associated pipeline disruptions, and 2) adjust the size of the fields used to score the similarity of the regions

[2] It is important to note that the envisioned hardware designs used in this estimate have not been implemented and may not produce the anticipate results while the cited report was obtained on actual hardware.

to reflect the precision required at each stage of the algorithm thus optimizing the movement of data between memory and registers.

7 Conclusion and future work

Programming multi-core parallel processors using conventional multitasking approaches is a difficult task. The associative data parallel paradigm promises to simplify that task, but several hardware modifications are necessary to effectively implement it. Specifically, 1) the core processors' vector operations need to be augmented with associative flag vectors to i) facilitate the coordination of vector operations on heterogeneous data, ii) reduce or eliminate the need for scatter-gather operations, iii) support dynamic data parallel memory allocation, iv) reduce the burden of partitioning tasks among multiple processors, v) support data parallel to scalar reduction and the insertion of scalar data into vectors; 2) a bit serial core is needed to support associative data parallel searching for the maximum, minimum, LUB, and GLB elements in a vector; 3) special hardware is needed to combine and partition the flag vectors when the logical vector to hardware vector register mappings vary due to the type of data being processed; 4) the flag vector needs to be augmented with a task ID field to support multitasking, then with an expanded fully capable control processor, multilevel heterogeneous parallelism can be supported.

The hardware modifications described here are primarily on a per core basis. Our future efforts will concentrate on 1) implementing associative support software on existing multi-core chips to verify our model, 2) determining what specific communication hardware enhancements are needed to support the associative model across the multiple cores of a chip, and 3) how the associative paradigm can be mapped across multiple multi-core chips.

8 References

[1] Falkoff, D., "Algorithms for Parallel-Search Memories," *J. ACM*, Vol. 9, 1962, pp. 488-511.

[2] Farrar, M., "Smith-Waterman Speeds Database Searches Six Times Over Other SIMD Implementations," *Bioinformatics*, Vol. 23, 2007, pp. 156-161.

[3] Flynn, M. J., "Very High-speed Computing Systems," *Proceedings of the IEEE*, Vol. 54, Dec. 1966, pp. 1901-1909.

[4] Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, Amsterdam, 2003.

[5] Hillis, W. D., *The Connection Machine*, MIT Press, 1989.

[6] *IBM Cell Broadband Engine Programming Handbook*, Version 1.0, April 19, 2006.

[7] Kogge, P. M., T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter, "Combined DRAM and Logic Chip for Massively Parallel Applications," *16th IEEE Conference on Advanced Research in VLSI*, 1995.

[8] Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review*, Vol. 63, 1956, pp. 81-97.

[9] Patterson, D., T. Anderson, N. Cardwell, R. Froman, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, Vol. 17, No. 2, 1997, pp. 34-44.

[10] Potter, J., J. Baker S. Scott, A. Bansal, C. Leangsuksun and C. Asthagiri, "ASC: An Associative Computing Paradigm," in *Associative Processing and Processors*, edited by A. Krikelis and C. Weems, IEEE Computer Society, Los Alamitos, CA, 1997, pp. 188-194.

[11] Potter, J., *Associative Computing – A Programming Paradigm for Massively Parallel Computers*, Plenum Publishing, New York, 1992.

[12] Potter, J. (ed.), *The Massively Parallel Processor*, MIT Press, 1985.

[13] Rognes, T. and E. Seeberg, "Six-fold Speed-up of the Smith-Waterman Sequence Database Searches Using Parallel Processing on Common Microprocessors," *Bioinformatics*, Vol. 16, 2000, pp. 699-706.

[14] Siegel, H. J., L. Wang, J. J. So, and M. Maheswaran, "Data Parallel Algorithms," in *Parallel and Distributed Computing Handbook*, edited by A. Y. Zomaya, McGraw-Hill, New York, NY, 1996, pp. 466-499.

[15] "Preparing for the Arrival of the Teraflops SP," *NPACI Online*, Vol. 3, Issue 8, April 14, www.npaci.edu/online/v3.8/SCAN1.html, 1999.