CAMBRIDGE TEXTS IN APPLIED MATHEMATICS

AN INTRODUCTION TO Parallel and Vector Scientific Computation

RONALD W. SHONKWILER
LEW LEFTON

CAMBRIDGE www.cambridge.org/9780521864787

This page intentionally left blank

An Introduction to Parallel and Vector Scientific Computing

In this text, students of applied mathematics, science, and engineering are introduced to fundamental ways of thinking about the broad context of parallelism. The authors begin by giving the reader a deeper understanding of the issues through a general examination of timing, data dependencies, and communication. These ideas are implemented with respect to shared memory, parallel and vector processing, and distributed memory cluster computing. Threads, OpenMP, and MPI are covered, along with code examples in Fortran, C, and Java.

The principles of parallel computation are applied throughout as the authors cover traditional topics in a first course in scientific computing. Building on the fundamentals of floating point representation and numerical error, a thorough treatment of numerical linear algebra and eigenvector/eigenvalue problems is provided. By studying how these algorithms parallelize, the reader is able to explore parallelism inherent in other computations, such as Monte Carlo methods.

Ronald W. Shonkwiler is a professor in the School of Mathematics at the Georgia Institute of Technology. He has authored or coauthored more than 50 research papers in areas of functional analysis, mathematical biology, image processing algorithms, fractal geometry, neural networks, and Monte Carlo optimization methods. His algorithm for monochrome image comparison is part of a U.S. patent for fractal image compression. He has coauthored two other books, *An Introduction to the Mathematics of Biology* and *The Handbook of Stochastic Analysis and Applications*.

Lew Lefton is the Director of Information Technology for the School of Mathematics and the College of Sciences at the Georgia Institute of Technology, where he has built and maintained several computing clusters that are used to implement parallel computations. Prior to that, he was a tenured faculty member in the Department of Mathematics at the University of New Orleans. He has authored or coauthored more than a dozen research papers in the areas of nonlinear differential equations and numerical analysis. His academic interests are in differential equations, applied mathematics, numerical analysis (in particular, finite element methods), and scientific computing.

Cambridge Texts in Applied Mathematics

All titles listed below can be obtained from good booksellers or from Cambridge University Press. For a complete series listing, visit http:publishing.cambridge.org/stm/mathematics/ctam

Thinking about Ordinary Differential Equations
Robert E. O'Malley Jr

A Modern Introduction to the Mathematical Theory of Water Waves
Robin Stanley Johnson

Rarefied Gas Dynamics: From Basic Concepts to Actual Calculations
Carlo Cercignani

Symmetry Methods for Differential Equations: A Beginner's Guide
Peter E. Hydon

High Speed Flow
C. J. Chapman

Wave Motion
J. Billingham, and A. C. King

An Introduction to Magnetohydrodynamics
P. A. Davidson

Linear Elastic Waves
John G. Harris

Vorticity and Incompressible Flow
Andrew J. Majda, and Andrea L. Bertozzi

Infinite-Dimensional Dynamical Systems
James C. Robinson

Introduction to Symmetry Analysis
Brain J. Cantwell

Bäcklund and Darboux Transformations
C. Rogers, and W. K. Schief

Finite Volume Methods for Hyperbolic Problems
Randall J. LeVeque

Introduction to Hydrodynamic Stability
P. G. Drazin

Theory of Vortex Sound
M. S. Howe

Scaling
Grigory Isaakovich Barenblatt

Complex Variables: Introduction and Applications (2nd Edition)
Mark J. Ablowitz, and Athanassios S. Fokas

A First Course in Combinatorial Optimization
Jone Lee

Practical Applied Mathematics: Modelling, Analysis, Approximation
Sam Howison

An Introduction to Parallel and Vector Scientific Computing

RONALD W. SHONKWILER
Georgia Institute of Technology

LEW LEFTON
Georgia Institute of Technology

CAMBRIDGE
UNIVERSITY PRESS

cambridge university press
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge cb2 2ru, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521864787

First published in print format 2006

isbn-13 978-0-511-24343-1 eBook (Adobe Reader)
isbn-10 0-511-24343-X eBook (Adobe Reader)

isbn-13 978-0-521-86478-7 hardback
isbn-10 0-521-86478-X hardback

isbn-13 978-0-521-68337-1 paperback
isbn-10 0-521-68337-8 paperback

Contents

Preface

Numerical computations are a fundamental tool for engineers and scientists. The current practice of science and engineering demands that nontrivial computations be performed with both great speed and great accuracy. More and more, one finds that scientific insight and technologial breakthroughs are preceded by intense computational efforts such as modeling and simulation. It is clear that computing is, and will continue to be, central to the further development of science and technology.

As market forces and technological breakthroughs lowered the cost of computational power by several orders of magintude, there was a natural migration from large-scale mainframes to powerful desktop workstations. Vector processing and parallelism became possible, and this parallelism gave rise to a new collection of algorithms. Parallel architectures matured, in part driven by the demand created by the algorithms. Large computational codes were modified to take advantage of these parallel supercomputers. Of course, the term *supercomputer* has referred, at various times, to radically different parallel architectures. This includes vector processors, various shared memory architectures, distributed memory clusters, and even computational grids. Although the landscape of scientific computing changes frequently, there is one constant; namely, that there will always be a demand in the research community for high-performance computing.

When computations are first introduced in beginning courses, they are often straightforward "vanilla" computations, which are well understood and easily done using standard techniques and/or commercial software packages on desktop computers. However, sooner or later, a working scientist or engineer will be faced with a problem that requires advanced techniques, more specialized software (perhaps coded from scratch), and/or more powerful hardware. This book is aimed at those individuals who are taking that step, from a novice to intermediate or even from intermediate to advanced user of tools that fall under

the broad heading of scientific computation. The text and exercises have been shown, over many years of classroom testing, to provide students with a solid foundation, which can be used to work on modern scientific computing problems. This book can be used as a guide for training the next generation of computational scientists.

This manuscript grew from a collection of lecture notes and exercises for a senior-level course entitled "Vector and Parallel Scientific Computing." This course runs yearly at the Georgia Institute of Technology, and it is listed in both mathematics and computer science curricula. The students are a mix of math majors, computer scientists, all kinds of engineers (aerospace, mechanical, electrical, etc.), and all kinds of scientists (chemists, physicists, computational biologists, etc.). The students who used these notes came from widely varying backgrounds and varying levels of expertise with regard to mathematics and computer science.

Formally, the prerequisite for using this text is knowledge of basic linear algebra. We integrate many advanced matrix and linear algebra concepts into the text as the topics arise rather than offering them as a separate chapter. The material in Part II, Monte Carlo Methods, also assumes some familiarity with basic probability and statistics (e.g., mean, variance, t test, Markov chains).

The students should have some experience with computer programming. We do not teach nor emphasize a specific programming language. Instead, we illustrate algorithms through a pseudocode, which is very close to mathematics itself. For example, the mathematical expression $y = \sum_{i=1}^{n} x_i$ becomes

```
y=0;
loop i = 1 upto n
    y = y + x_i;
end loop
```

We provide many example programs in Fortran, C, and Java. We also have examples of code that uses MPI libraries. When this course was originally taught, it took several weeks for the students to get accounts and access to the Cray system available at that time. As a result, the material in the first two chapters provides no programming exercises. If one wishes to start programming right away, then he or she should begin with Chapter 3.

The purpose of the course is to provide an introduction to important topics of scientific computing including the central algorithms for numerical linear algebra such as linear system solving and eigenvalue calculation. Moreover, we introduce this material from the very beginning in the context of vector and parallel computation. We emphasize a recognition of the sources and propagation of numerical error and techniques for its control. Numerical error starts with

the limitations inherent in the floating point representation of numbers leading to round-off error and continues with algorithmic sources of error.

The material has evolved over time along with the machines called supercomputers. At present, shared memory parallel computation has standardized on the threads model, and vector computation has moved from the machine level to the chip level. Of course, vendors provide parallelizing compilers that primarily automatically parallelize loops that the programmer has requested, sometimes referred to as the DOACROSS model. This is a convenient model for engineers and scientists as it allows them to take advantage of parallel and vector machines while making minimal demands on their programming time. For the purpose of familiarily, we include a section on the basic concepts of distributed memory computation, including topological connectivity and communication issues.

In teaching the course, we employ a hands-on approach, requiring the students to write and execute programs on a regular basis. Over the years, our students have had time on a wide variety of supercomputers, first at National Centers, and more recently at campus centers or even on departmental machines. Of course, even personal computers today can be multiprocessor with a vector processing chipset, and many compiled codes implement threads at the operating system level.

We base our approach to parallel computation on its representation by means of a directed acyclic graph. This cuts to the essence of the computation and clearly shows its parallel structure. From the graph it is easy to explain and calculate the complexity, speedup, efficiency, communication requirements, and scheduling of the computation. And, of course, the graph shows how the computation can be coded in parallel.

The text begins with an introduction and some basic terminology in Chapter 1. Chapter 2 gives a high-level view of the theoretical underpinnings of parallelism. Here we discuss data dependencies and complexity, using directed acyclic graphs to more carefully demonstate a general way of thinking about parallelism. In Chapter 3, we have included a variety of machine implementations of parallelism. Although some of these architectures are not in widespread use any more (e.g., vector processors like the early Cray computers), there are still interesting and important ideas here. In fact, the Japanese Earth Simulator (the former world record holder for "fastest computer") makes heavy use of vector processing and pipelining. Chapter 3 includes an introduction to low-level implementations of parallelism by including material on barriers, mutexes, and threads. Of course, not every scientific computing application will require thread programming, but as mentioned earlier, these objects provide many useful ideas about parallelization that can be generalized to many different parallel codes.

We have even included a short introduction to quantum computing because this technology may one day be the future of parallel scientific computation.

In the second half of the book, we start with basic mathematical and computational background, presented as building blocks in Chapter 4. This includes material on floating point numbers, round-off error, and basic matrix arithmetic. We proceed to cover mathematical algorithms, which we have found are most frequently used in scientific computing. Naturally, this includes a large measure of numerical linear algebra. Chapters 5, 6, and 7 discuss direct methods for solving linear systems. We begin with classical Gaussian elimination and then move on to matrices with special structure and more advanced topics such as Cholesky decomposition and Givens' rotation. Iterative methods are covered in Chapter 8. We study Jacobi and Gauss-Seidel as well as relaxtion techniques. This chapter also includes a section on conjugate gradient methods. In Chapter 9, we examine eigenvalues and eigenvectors. This includes the power method and QR decomposition. We also cover the topics of Householder transformations and Hessenberg forms, since these can improve QR computations in practice.

Throughout all of Part II, our development of linear algebraic results relies heavily on the technique of partitioning matrices. This is introduced in Chapter 4 and continues through our presentation of Jordan form in Chapter 9.

The final section of the book is focused on Monte Carlo methods. We first develop classical quadrature techniques such as the Buffon Needle Problem in Chapter 10. We then advance in Chapter 11 to a presentation of Monte Carlo optimization, which touches on the ideas of simulated annealing, genetic algorithms, and iterated improvement with random restart.

Exercises are included at the end of every section. Some of these are meant to be done by hand, and some will require access to a computing environment that supports the necessary parallel architecture. This could be a vector machine, an SMP system supporting POSIX threads, a distributed memory cluster with MPI libraries and compilers, etc. We have attempted to isolate those exercises that require programming in a subsection of each exercise set. Exercises are followed by a number in parentheses, which is meant to be an indication of the level of difficulty.

Because scientific computing is often the result of significant research efforts by large distributed teams, it can be difficult to isolate meaningful self-contained exercises for a textbook such as this. We have found it very useful for students to work on and present a project as a substantial part of their course grade. A 10-minute oral presentation along with a written report (and/or a poster) is an excellent exercise for students at this level. One can ask them to submit a short project proposal in which they briefly describe the problem background, the

mathematical problem that requires computation, and how this computation may parallelize. Students do well when given the opportunity to perform a deeper study of a problem of interest to them.

Acknowledgments

Ron Shonkwiler would like to thank his coauthor, Dr. Lew Lefton, for bringing thoughtfulness and a fresh outlook to this project and for his "way with words," and Lauren Cowles of Cambridge for her encouragement. Most of all, he would like to thank the many students who have taken the course on which this book is based. Their enthusiasm, creativity, energy, and industry have made teaching the course a challenge and a pleasure.

Lew Lefton would first and foremost like to express his deepest gratitude to his coauthor, Dr. Ron Shonkwiler. When they began collaborating on the project of getting the course notes into a book manuscript several years ago, Lefton naively thought it would be an interesting but straightforward project. He was half right; it has been very interesting. The notes have changed (improved!) signifcantly since that early draft and, having taught the course himself and gone through the process of publishing a book, he feels very fortunate to have had Ron, with his expertise and experience, as a guide. Lefton would also like to thank his wife, Dr. Enid Steinbart, and his daughters, Hannah, Monica, and Natalie, who put up with him spending more than his usual amount of "screen time" in front of the computer. He is truly grateful for their love and support.

PART I

Machines and Computation

1

Introduction – The Nature of High-Performance Computation

The need for speed. Since the beginning of the era of the modern digital computer in the early 1940s, computing power has increased at an exponential rate (see Fig. 1). Such an exponential growth is predicted by the well-known "Moore's Law," first advanced in 1965 by Gordon Moore of Intel, asserting that the number of transistors per inch on integrated circuits will double every 18 months. Clearly there has been a great need for ever more computation. This need continues today unabated. The calculations performed by those original computers were in the fields of ballistics, nuclear fission, and cryptography. And, today these fields, in the form of computational fluid dynamics, advanced simulation for nuclear testing, and cryptography, are among computing's Grand Challenges.

In 1991, the U.S. Congress passed the High Performance Computing Act, which authorized The Federal High Performance Computing and Communications (HPCC) Program. A class of problems developed in conjunction with the HPCC Program was designated "Grand Challenge Problems" by Dr. Ken Wilson of Cornell University. These problems were characterized as "fundamental problems in science and engineering that have broad economic or scientific impact and whose solution can be advanced by applying high performance computing techniques and resources." Since then various scientific and engineering committees and governmental agencies have added problems to the original list. As a result, today there are many Grand Challenge problems in engineering, mathematics, and all the fundamental sciences. The ambitious goals of recent Grand Challenge efforts strive to

- build more energy-efficient cars and airplanes,
- design better drugs,
- forecast weather and predict global climate change,
- improve environmental modeling,

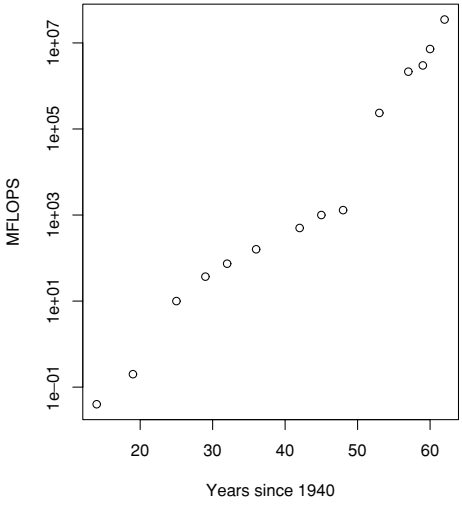

Fig. 1. Computational speed in MFLOPS vs. year.

- improve military systems,
- understand how galaxies are formed,
- understand the nature of new materials, and
- understand the structure of biological molecules.

The advent of high-speed computation has even given rise to computational subfields in some areas of science and engineering. Examples are computational biology, bioinfomatics, and robotics, just to name a few. Computational chemistry can boast that in 1998 the Noble Prize in chemistry was awarded to John Pope and shared with Walter Kohn for the development of computational methods in quantum chemistry.

And so it seems that the more computational power we have, the more use we make of it and the more we glimpse the possibilities of even greater computing power. The situation is like a Moore's Law for visionary computation.

1.1 Computing Hardware Has Undergone Vast Improvement

A major factor in the exponential improvement in computational power over the past several decades has been through advances in solid-state physics: faster switching circuits, better heat control, faster clock rates, faster memory. Along with advances in solid-state physics, there has also been an evolution in the architecture of the computer itself. Much of this revolution was spearheaded by Seymour Cray.

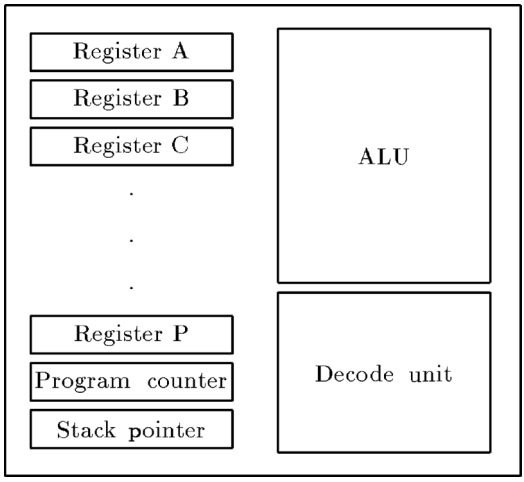

Fig. 2. Central processing unit.

Many ideas for parallel architectures have been tried, tested, and mostly discarded or rethought. However, something is learned with each new attempt, and the successes are incorporated into the next generation of designs. Ideas such as interleaved memory, cache memory, instruction look ahead, segmentation and multiple functional units, instruction piplining, data pipelining, multiprocessing, shared memory, distributed memory have found their way into the various catagories of parallel computers available today. Some of these can be incorporated into all computers, such as instruction look ahead. Others define the type of computer; thus, vector computers are data pipelined machines.

The von Neumann Computer

For our purposes here, a computer consists of a central processing unit or *CPU*, memory for information storage, a path or *bus* over which data flow and a synchronization mechanism in the form of a *clock*. The CPU itself consists of several internal registers – a kind of high-speed memory, a program counter (PC), a stack pointer (SP), a decode unit (DU), and an arithmetic and logic unit (ALU) (see Fig. 2). A program consists of one or more contiguous memory locations, that is, chunks of memory, containing a *code segment* including subroutines, a *data segment* for the variables and parameters of the problem, a *stack segment*, and possibly additional memory allocated to the program at run time (see Fig. 3).

The various hardware elements are synchronized by the clock whose frequency f characterizes the speed at which instructions are executed. The

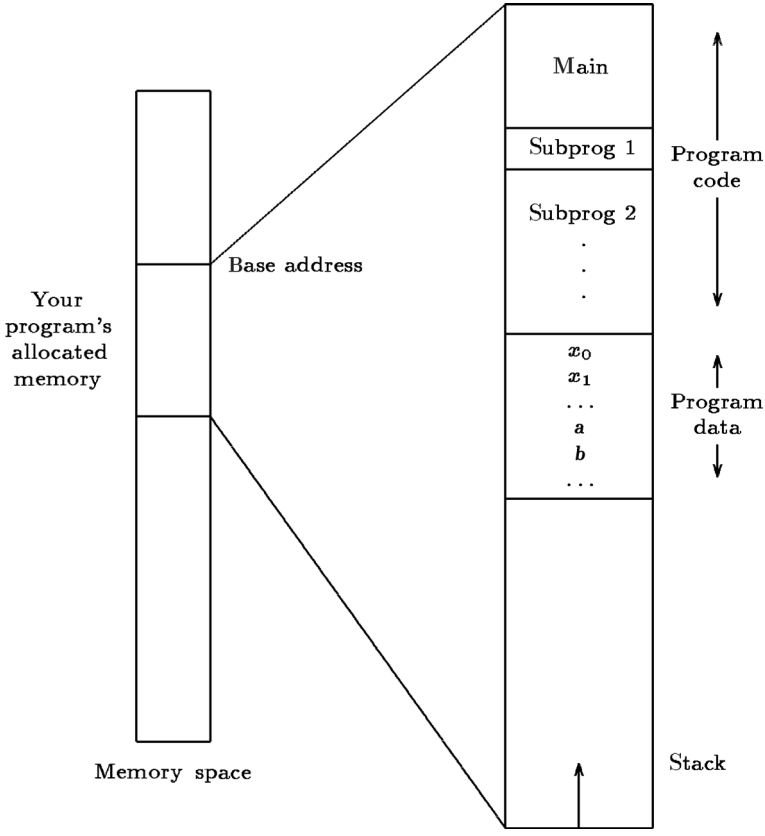

Fig. 3. Organization of main memory.

frequency is the number of cycles of the clock per second measured in megaHertz (mHz), 1 mHz $= 10^6$ Hz or gigaHertz, (gHz), 1 gHz $= 10^9$ Hz. The time t for one clock cycle is the reciprocal of the frequency

$$t = \frac{1}{f}.$$

Thus a 2-ns clock cycle corresponds to a frequency of 500 mHz since 1 ns $=$ 10^{-9} s and

$$f = \frac{1}{2 \times 10^{-9}} = 0.5 \times 10^9 = 500 \times 10^6.$$

If one instruction is completed per clock cycle, then the instruction rate, *IPS*, is the same as the frequency. The instruction rate is often given in millions of

instructions per second or *MIPS*; hence, *MIPS* equals megaHertz for such a computer.

The original computer architecture, named after John von Neumann, who was the first to envision "stored programming" whereby the computer could change its own course of action, reads instructions one at a time sequentially and acts upon data items in the same way. To gain some idea of how a von Neumann computer works, we examine a step-by-step walk-through of the computation $c = a + b$.

Operation of a von Neumann Computer: $c = a + b$ Walk-Through

On successive clock cycles:

Step 1. Get next instruction
Step 2. Decode: fetch a
Step 3. Fetch a to internal register
Step 4. Get next instruction
Step 5. Decode: fetch b
Step 6. Fetch b to internal register
Step 7. Get next instruction
Step 8. Decode: add a and b (result c to internal register)
Step 9. Do the addition in the ALU (see below)
Step 10. Get next instruction
Step 11. Decode: store c (in main memory)
Step 12. Move c from internal register to main memory

In this example two *floating point number*s are added. A floating point number is a number that is stored in the computer in mantissa and exponent form (see Section 4.1); integer numbers are stored directly, that is, with all mantissa and no exponent. Often in scientific computation the results materialize after a certain number of *floating point operation*s occur, that is, additions, subtractions, multiplications, or divisions. Hence computers can be rated according to how many floating point operations per second, or FLOPS, they can perform. Usually it is a very large number and hence measured in mega-FLOPS, written MFLOPS, or giga-FLOPS written GFLOPS, or tera-FLOPS (TFLOPS). Of course, 1 MFLOPS $= 10^6$ FLOPS, 1 GFLOPS $= 10^3$ MFLOPS $= 10^9$ FLOPS, and 1 TFLOPS $= 10^{12}$ FLOPS.

The addition done at step 9 in the above walk-through consists of several steps itself. For this illustration, assume 0.9817×10^3 is to be added to 0.4151×10^2.

Step 1. Unpack operands: 9817 | 3 4151 | 2
Step 2. Exponent compare: 3 vs. 2

Step 3. Mantissa align: 9817 | 3 0415 | 3
Step 4. Mantissa addition: 10232 | 3
Step 5. Normalization (carry) check: $\boxed{1}$0232 | 3
Step 6. Mantissa shift: 1023 | 3
Step 7. Exponent adjust: 1023 | 4
Step 8. Repack result: 0.1023×10^4

So if the clock speed is doubled, then each computer instruction takes place in one half the time and execution speed is doubled. But physical laws limit the improvement that will be possible this way. Furthermore, as the physical limits are approached, improvements will become very costly. Fortunately there is another possibility for speeding up computations, parallelizing them.

Parallel Computing Hardware – Flynn's Classification

An early attempt to classify parallel computation made by Flynn is somewhat imprecise today but is nevertheless widely used.

	Single-data stream	Multiple-data streams
Single instruction	von Neumann	SIMD
Multiple instructions		MIMD

As we saw above, the original computer architecture, the von Neumann computer, reads instructions one at a time sequentially and acts upon data in the same way; thus, they are single instruction, single data, or SISD machines.

An early idea for parallelization, especially for scientific and engineering programming, has been the vector computer. Here it is often the case that the same instruction is performed on many data items as if these data were a single unit, a mathematical vector. For example, the scalar multiplication of a vector multiplies each component by the same number. Thus a single instruction is carried out on multiple data so these are SIMD machines. In these machines the parallelism is very structured and fine-grained (see Section 1.3).

Another term for this kind of computation is *data parallelism*. The parallelism stems from the data while the program itself is entirely serial. Mapping each instruction of the program to its target data is done by the compiler. Vector compilers automatically parallelize vector operations, provided the calculation is *vectorizable*, that is, can be correctly done in parallel (see Section 3.6).

Modern languages incorporate special instructions to help the compiler with the data partitioning. For example, the following statements in High Performance Fortran (HPF)

```
      real x(1000)
!HPF$ PROCESSORS p(10)
!HPF$ DISTRIBUTE x(BLOCK) ONTO p
```

invokes 10 processors and instructs the 1,000 elements of x to be distributed with $1{,}000/10 = 100$ contiguous elements going to each.

Another approach to SIMD/data partitioned computing, massively parallel SIMD, is exemplified by the now extinct Connection Machine. Here instructions are broadcast (electronically) to thousands of processors each of which is working on its own data.

True, flexible, parallel computation comes about with multiple independent processors executing, possibly different, instructions at the same time on different data, that is, multiple instruction multiple data or MIMD computers. This class is further categorized according to how the memory is configured with respect to the processors, centralized, and shared or distributed according to some topology.

We consider each of these in more detail below.

1.2 SIMD–Vector Computers

In the von Neumann model, much of the computer hardware is idle while other parts of the machine are working. Thus the Decode Unit is idle while the ALU is calculating an addition for example. The idea here is to keep all the hardware of the computer working all the time. This is parallelism at the hardware level.

Operation of a Vector Computer – Assembly-Line Processing

First the computer's hardware is modularized or *segmented* into functional units that perform well-defined specialized tasks (see, for example, the Cray architecture diagram Fig. 16). The vector pipes are likewise segmented. Figure 4 shows the segments for the Cray add pipe.

It is desirable that the individual units be as independent as possible. This idea is similar to the modularization of an assembly plant into stations each of which performs a very specific single task. Like a factory, the various detailed steps of processing done to the code and data of a program are formalized, and specialized hardware is designed to perform each such step at the same time as

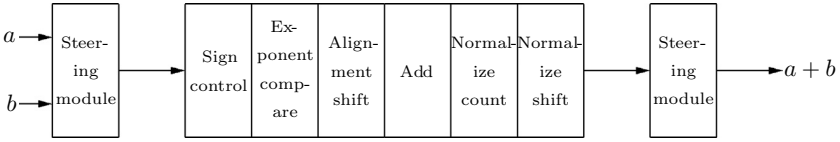

Fig. 4. A block diagram of the Cray add unit.

all the other steps. Then the data or code is processed step by step by moving from segment to segment; this is *pipelining*.

In our model of a computer, some of the main units are the fetch and store unit, the decode unit, and the arithmetic and logic unit. This makes it possible, for example, for the instructions of the program to be fetched before their turn in the execution sequence and held in special registers. This is called *caching*, allowing for advance decoding. In this way, operands can be prefetched so as to be available at the moment needed. Among the tasks of the decode unit is to precalculate the possible branches of conditional statements so that no matter which branch is taken, the right machine instruction is waiting in the instruction cache.

The innovation that gives a vector computer its name is the application of this principle to floating point numerical calculations. The result is an assembly line processing of much of the program's calculations. The assembly line in this case is called a *vector pipe*.

Assembly line processing is effective especially for the floating point operations of a program. Consider the sum of two vectors $\mathbf{x} + \mathbf{y}$ of length 200. To produce the first sum, $x_1 + y_1$, several machine cycles are required as we saw above. By analogy, the first item to roll off an assembly line takes the full time required for assembling one item. But immediately behind it is the second item and behind that the third and so on. In the same way, the second and subsequent sums $x_i + y_i$, $i = 2, \ldots, 200$, are produced one per clock cycle. In the next section we derive some equations governing such vector computations.

Example. Calculate $y_i = x_i + x_i^2$ for $i = 1, 2, \ldots, 100$

```
loop i = 1...100
    yi = xi*(1+xi)    or?    yi = xi + xi * xi
end loop
```

Not all operations on mathematical vectors can be done via the vector pipes. We regard a *vector operation* as one which can. Mathematically it is an operation on the components of a vector which also results in a vector. For example, vector addition $\mathbf{x} + \mathbf{y}$ as above. In components this is $z_i = x_i + y_i$, $i = 1, \ldots, n$, and would be coded as a loop with index i running from 1 to n. Multiplying two

Table 1. *Vector timing data*[*]

Type of arithmetic operation	Time in ns for n operations
Vector add/multiply/boolean	$1000 + 10n$
Vector division	$1600 + 70n$
Saxpy (cf. pp 14)	$1600 + 10n$
Scalar operation[**]	$100n$
Inner product	$2000 + 20n$
Square roots	$500n$

[*] For a mid-80's memory-to-memory vector computer.
[**] Except division, assume division is 7 times longer.

vectors componentwise and scalar multiplication, that is, the multiplication of the components of a vector by a constant, are other examples of a vector operation.

By contrast, the inner or dot product of two vectors is not a vector operation in this regard, because the requirement of summing the resulting componentwise products cannot be done using the vector pipes. (At least not directly, see the exercises for pseudo-vectorizing such an operaton.)

Hockney's Formulas

Let t_n be the time to calculate a vector operation on vectors of length of n. If s is the number of clock cycles to prepare the pipe and fetch the operands and l is the number of cycles to fill up the pipe, then $(s + l)\tau$ is the time for the first result to emerge from the pipe where τ is the time for a clock cycle. Thereafter, another result is produced per clock cycle, hence

$$t_n = (s + l + (n - 1))\tau,$$

see Table 1.

The *startup time* is $(s + l - 1)\tau$ in seconds. And the *operation rate*, r, is defined as the number of operations per unit time so

$$r = \frac{n}{t_n}.$$

Theoretical peak performance, r_∞, is one per clock cycle or

$$r_\infty = \frac{1}{\tau}.$$

Thus we can write

$$r = \frac{r_\infty}{1 + \frac{s+l-1}{n}}.$$

This relationship is shown in Fig. 5.

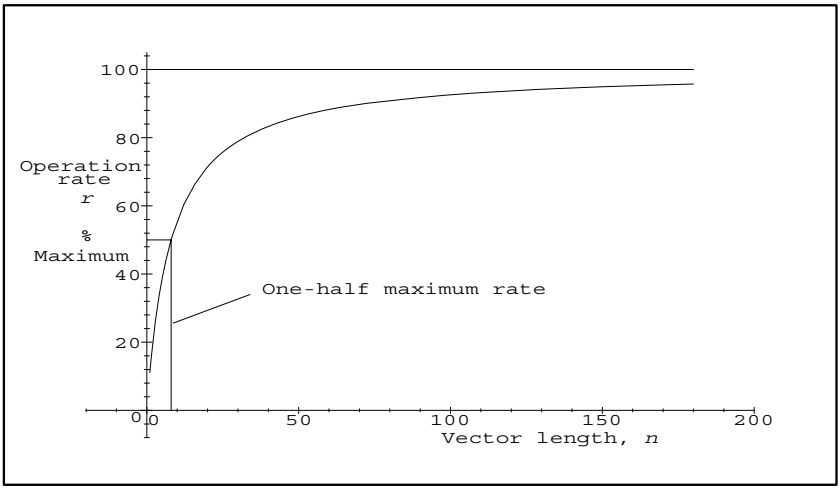

Fig. 5. Operation rate vs. vector length.

Hockney's $n_{1/2}$ value is defined as the vector length for achieving one-half peak performance, that is,

$$\frac{1}{2\tau} = \frac{n_{1/2}}{(s + l - 1 + n_{1/2})\tau}.$$

This gives

$$n_{1/2} = s + l - 1$$

or equal to the startup time. Using $n_{1/2}$, the operation rate can now be written

$$r = \frac{r_\infty}{1 + \frac{n_{1/2}}{n}}.$$

Hockney's *break-even point* is defined as the vector length for which the scalar calculation takes the same time as the vector calculation. Letting $r_{\infty,v}$ denote the peak performance in vector mode and $r_{\infty,s}$ the same in scalar mode, we have

$$\text{vector time for } n = \text{scalar time for } n$$
$$\frac{s + l - 1 + n_b}{r_{\infty,v}} = \frac{n_b}{r_{\infty,s}}.$$

Solving this for n_b gives

$$n_b = \frac{n_{1/2}}{\frac{r_{\infty,v}}{r_{\infty,s}} - 1}.$$

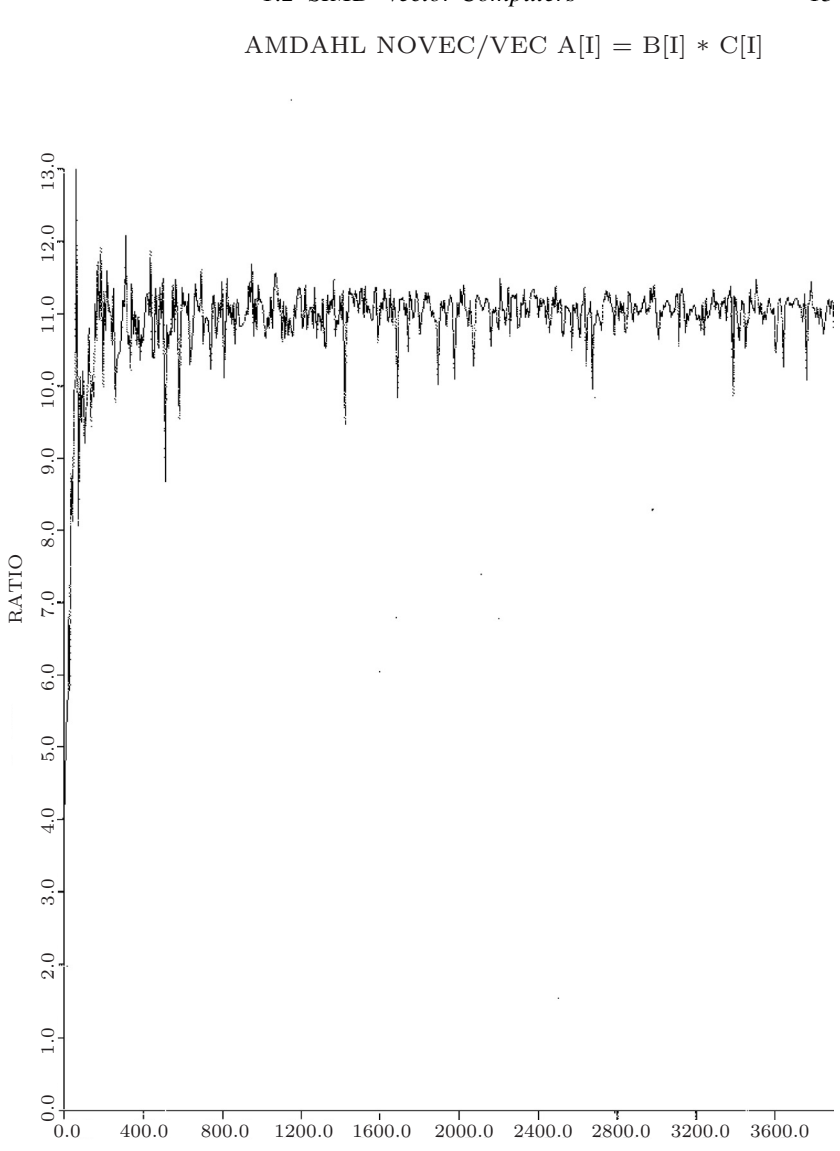

Fig. 6. Gain for vector multiplication.

This calculation is not exactly correct for a register machine such as the Cray since, by having to shuttle data between the vector registers and the vector pipes, there results a bottleneck at the vector registers. Of course, $r_{\infty,v} > r_{\infty,s}$ (or else there is no need for a vector computer). At one time that ratio was about 4 for Cray machines and n_b was about 8.