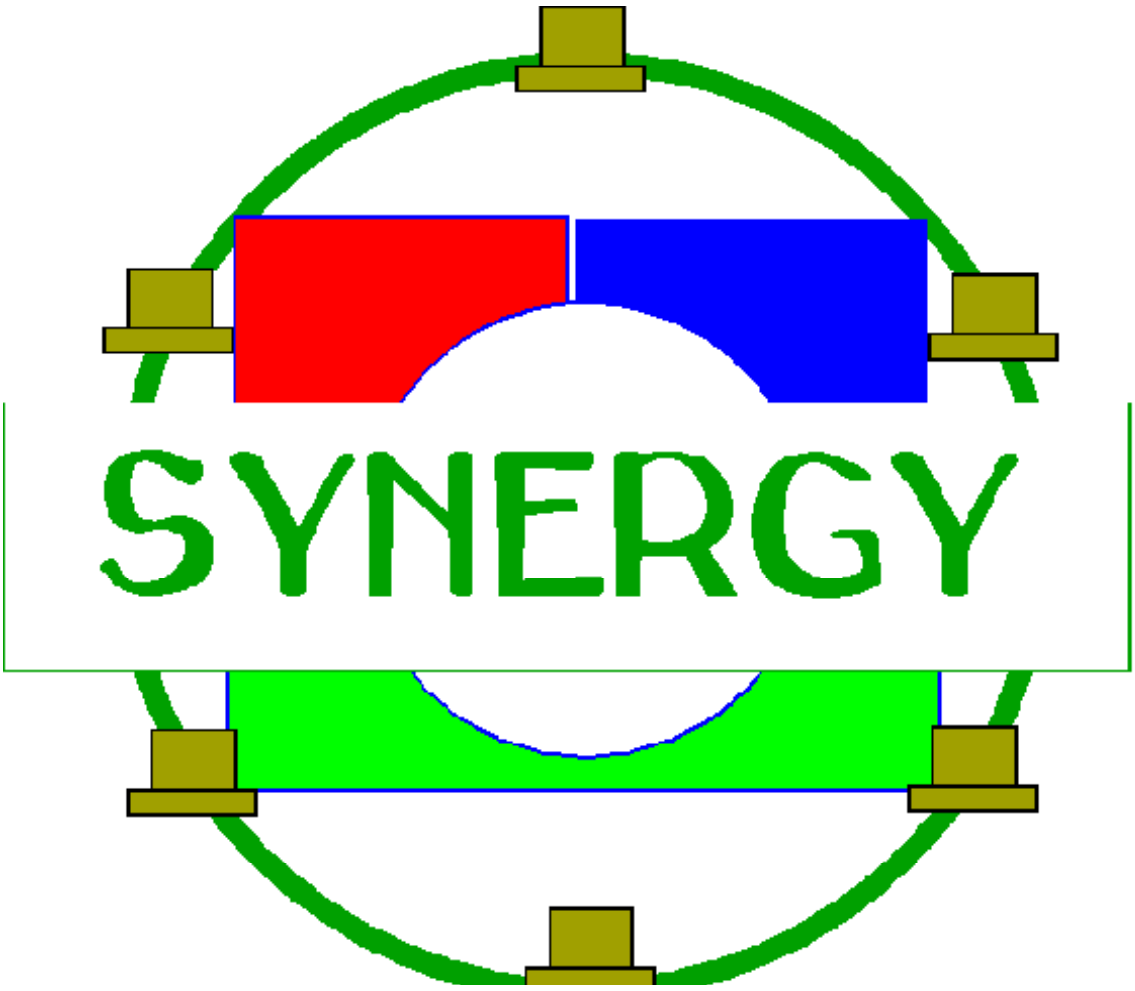


Synergy User Manual And Tutorial

Documenting the Synergy Project

Supervised by Dr. Yuan Shi
Compiled by Joe Jupin

syn·er·gy (sĭn ər-jē) *noun*
plural **syn·er·gies**

1. The interaction of two or more agents or forces so that their combined effect is greater than the sum of their individual effects.
2. Cooperative interaction among groups, especially among the acquired subsidiaries or merged parts of a corporation, that creates an enhanced combined effect.

[From Greek *sunergia*, cooperation, from *sunergos*, working together.]

"For it is unworthy of excellent men to lose hours like slaves in the labour of calculation which could safely be relegated to anyone else if machines were used."

-Gottfried Wilhelm Leibniz

Table of Contents

Introduction

Red text: Copied and pasted from syng_man.ps by Dr. Shi

The emergence of low cost, high performance uni-processors forces the enlargement of processing grains in all multi-processor systems. Consequently, individual parallel programs have increased in length and complexities. However, like reliability, parallel processing of any multiple communicating sequential programs is not really a functional requirement.

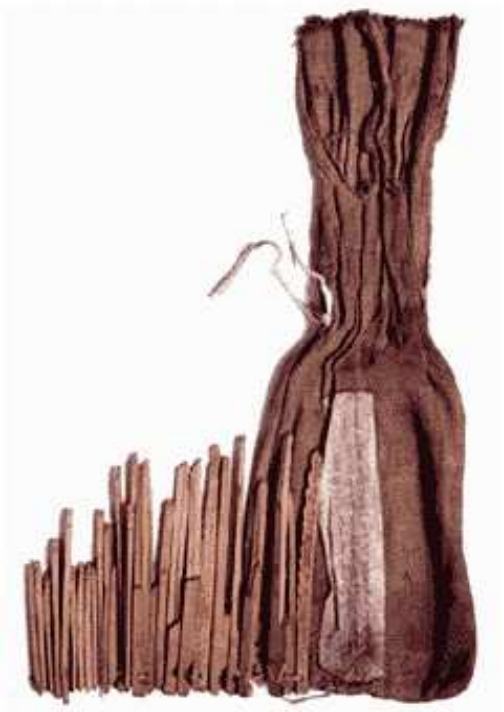
Separating pure functional programming concerns from parallel processing and resource management concerns can greatly simplify the conventional ``parallel programming'' asks. For example, the use of dataflow principles can facilitate automatic task scheduling. Smart tools can automate resource management. As long as the application dependent parallel structure is uncovered properly, we can even automatically assign processors to parallel programs in all cases.

Synergy V3.0 is an implementation of above ideas. It supports parallel processing using multiple ``Unix computers'' mounted on multiple file systems (or clusters) using TCP/IP. It allows parallel processing of any application using mixed languages, including parallel programming languages. Synergy may be thought of as a successor to Linda[1], PVM[2] and Express[3].

Our need to store and process data has been continually increasing for thousands of years. This need has lead to the development of complex storage, communication, numerical and processing systems. The information in this section was wholly obtained from sources freely available on the Internet, which are cited in the references section. Much of it was obtained from timelines, encyclopedias and academic Web pages. The accuracy of information collected from the Internet was checked by using multiple corroborating resources and eliminating contradictory information.

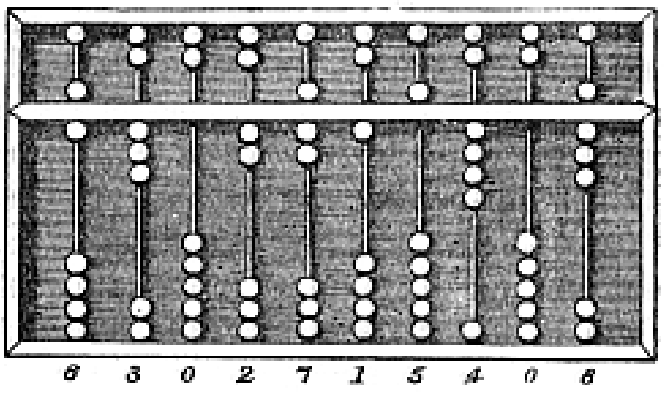
[1] Linda is a tuple space parallel programming system lead by Dr. David Gelenter, Yale University. Its commercial version is distributed by the Scientific Computing Associates, New Heaven, NH.
[2] PVM is a message passing parallel programming system by Oak Ridge National Laboratory, University of Tennessee and Emory University.
[3] Express is a commercial message passing parallel programming system by ParaSoft, CA.

History and Limitations of Ancient and Traditional Computing



The first recognized use of a tool to record the result of transactions was a device called a tally stick. The oldest known artifact is a wolf bone with a series of fifty-five cuts in groups of five that dates from approximately 30,000 to 25,000 BC. The notches in the stick may refer to the number of coins or other items that are counted by some early form of bookkeeping. The earliest stock markets used tally sticks to record transactions. The word "stock" actually means a stout stick. During a transaction the "broker" would record the transaction for the purchase of stock on a tally stick and then "break" the stick, keeping half and giving the other half to the investor. The two halves would be fit together at some later time to verify the investor's ownership of the shares of stock. In 1734 the English government ordered the cessation of the use of tally sticks but they were not completely abolished until 1826. By 1834 British Parliament collected a very large number of tally sticks, which the decided to burn in the fireplace at the House of Lords. The fireplace was "engorged" with tally sticks such that the fire spread to the paneling and to the neighboring House of Commons, destroying both buildings, which took ten years to reconstruct.[1] Other primitive recording devices included clay tablets, knotted strings, pebbles in bags and parchments. In modern times, books or legers have been used to record commercial or financial data using more formal bookkeeping systems, such as the double entry standard that is widely used today.

The first place-valued numerical system, in which both digit and position within the number determine value, and the abacus, which was the first actual calculating mechanism, are believed to have been invented by the Babylonians sometime between 3000 and 500 BC. Their number system is believed to have been developed based on astrological observations. It was a sexagesimal (base-60) system, which had the advantage of being wholly divisible by 2, 3, 4, 5, 6, 10, 15, 20 and 30. The first abacus was likely a stone covered with sand on which pebbles were moved across lines drawn in the sand. Later improvements were constructed from wood frames with either thin sticks or a tether material on which clay beads or pebbles were threaded. Sometime between



200 BC the 14th century, the Chinese invented a more advanced abacus device. The typical Chinese swanpan (abacus) is approximately eight inches tall and of various widths and typically has more than seven rods, which hold beads usually made from hardwood. This device works as a 5-2-5-2 based number system, which is similar to the decimal system. Advanced swanpan techniques are not limited to simple addition and subtraction. Multiplication, division, square roots and cube roots can be calculated very efficiently. A variation of this devise is still in use by shopkeepers in various Asian countries.[ii] There is direct evidence that the Chinese were using a positional number system by 1300 BC and were using a zero valued digit by 800 AD.

Sometime after 200 BC, Eratosthenes of Cyrene (276-194 BC) developed the Sieve of Eratosthenes, which was a procedure for determining prime numbers. It is called a sieve because it strains or filters out all non-primes. The process is as follows:

1. Make a list of all integers greater than one and less than or equal to n
2. Strike out the multiples of all primes less than or equal to the square root of n.
3. The numbers that are left are the primes.

The table below show the result for n = 50 with primes in the white squares.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Eratosthenes is also credited with being the first person to accurately estimate the diameter of the Earth and also served as the director of the famed Library of Alexandria.[iii]

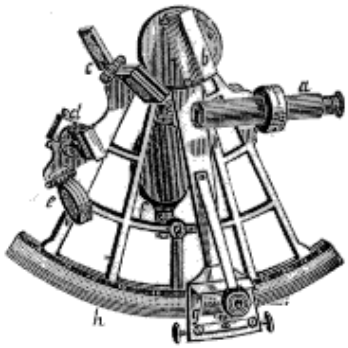
A postage stamp issued by the USSR in 1983 to commemorate the 1200th anniversary of Muhammad al-Khowarizmi. Scanned by Donald Knuth, one of the legends of computer science.

The Sieve of Eratosthenes is one of the first well-documented uses of an efficient algorithm-type solution to solve a complex problem. The word algorithm is derived from the Latin derivation of Al-Khowarizmi's name. Muhammad ibn Musa al-Khwarizmi was an Arab mathematician of the court of Mamun in Baghdad born before 800 AD in central Asia, now called Uzbekistan. Along with other Arabic mathematicians, he is responsible for the proliferation of the base-ten number system, which was developed in India. His book on the subject of Hindu numerals was later translated into the Latin text Liber Algorismi de numero Indorum. While a scholar at the House of Wisdom in Baghdad, he wrote Hisãb al-jabr w'al-muqãbala (from which the word "algebra" is derived). Lose translations of this title could be "the science of transposition and cancellation" or "the calculation of reduction and restoration." He devised a method to restore or transpose negative terms to the other side of an equation and reduce (cancel) or unite similar terms on either side of the equation. Transposition means that a quantity can be added or subtracted (multiplied or divided) from both sides of an equation and cancellation means that if there are two equal terms on either side of an equation, they can be altogether cancelled. The following is a translation of a popular verse in Arab schools from over six hundred years ago:

> *Cancel minus terms and then*
> *Restore to make your algebra;*
> *Combine your homogeneous terms*
> *And this is called muqabalah.*

Robert of Chester translated this work into Latin in 1140 AD. Similar methods are still in use in modern algebraic manipulations, which came in the sixteenth century from Francois Viète. Al-Khowarizmi also claimed in his book Indorum (the book of Al-Khowarizmi) that any complex mathematical problem could be broken down into smaller, simpler sub-problems, whose results could be logically combined to solve the initial problem. This is the main concept of an algorithm. Latin translations of his work contributed to much of medieval Europe's knowledge of mathematics. In 1202, Leonardo of Pisa (otherwise known by his nickname Fibonacci) (c. 1175-1250) wrote the

historic book Liber Abaci or “The Book of Calculation”, which was his interpretation of the Arabic-Hindu decimal number system that he learned while traveling with Arabs in North Africa. This book was the first to expose the general public, rather than academia, to the decimal number system, which quickly gained popularity because of its clear superiority over existing systems. [iv]

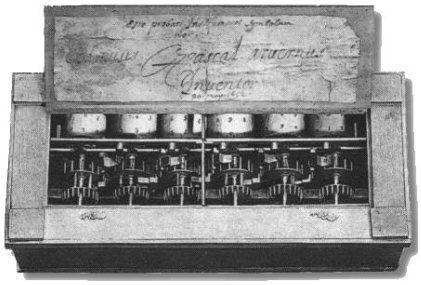


Sextant, p. 1932.

The Greek astronomer, geographer and mathematician Hipparchus (c. 190 BC – 120 BC) likely invented the navigational instrument called an astrolabe. This is a protractor-like device consisting of a degree marked circle with a center attached rotating arm. When the zero degree mark is aligned on the horizon and a celestial body is sighted along the movable arm, the celestial body’s position can be read from the degree marks on the circle. The sextant eventually replaced this device because the sextant measured relative to the horizon and not the device itself, which allowed more accurate measurements of position for latitude.

Sometime between 1612 and 1614, John Napier (1550 - 1617), born at Merchiston Tower in Edinburgh, Scotland, developed the decimal point, logarithms and Napier’s bones—an abacus for the calculation of products and quotients of numbers. Hand performed calculations were made much easier by the use of logarithms, which made possible many later scientific advancements. Mirifici Logarithmorum Canonis Descriptio or in English "Description of the Marvelous Canon of Logarithms", his mathematical work, contained thirty-seven pages of explanatory matter and ninety pages of tables, which furthered advancements in astronomy, dynamics and physics. Based on Napier’s algorithms in 1622, William Oughtred (1574 - 1660) invented the circular slide rule for calculating multiplication and division. In 1632 he published Circles of Proportion and the Horizontal Instrument, which described slide rules and sundials. By 1650 the sliding stick form of the slide rule was developed. In

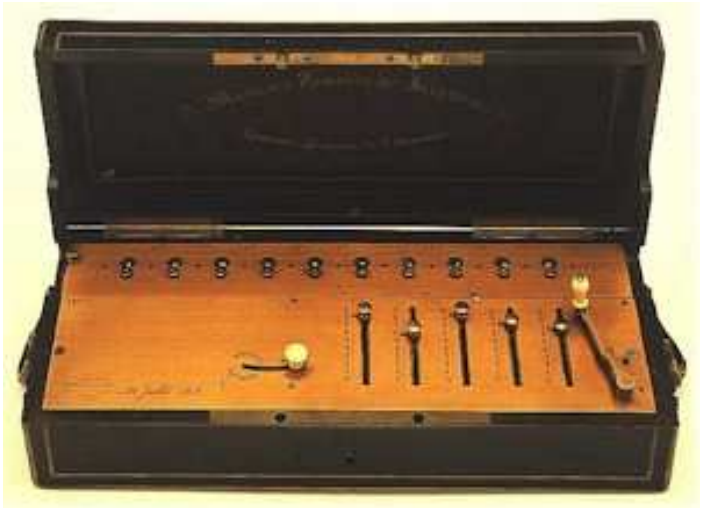
1624, Henry Briggs (1561 - 1630) published the first set of modern logarithms, and in 1628, Adrian Vlacq published the first complete set of modern logarithms.



In 1623, Wilhelm Schickard (1592 - 1635) invented what is believed to be the first mechanical calculating machine (left). This device used a "calculating clock" with a gear driven carry for mechanism to calculate the multiplication of multi-digit numbers in higher order positions. Between 1642 and 1643, at the age of 18, Blaise Pascal (1623 - 1662) created the "Pascaline" (right) a gear driven adding machine, which was the first mechanical adding/subtracting machine. Pascal developed this machine to help his father with his work—a tax collector. He discovered how to mechanically carry numbers to the next high order by causing the higher order gear to advance one tooth for a full rotation (ten teeth) of the next lower ordered gear. This method is similar to that of old pinball machines or gas pumps with rotating number counters. These devices were never placed into commercial service due to high cost of manufacture. Approximately fifty Pascalines were constructed and could handle calculations with up to eight digits.[v]

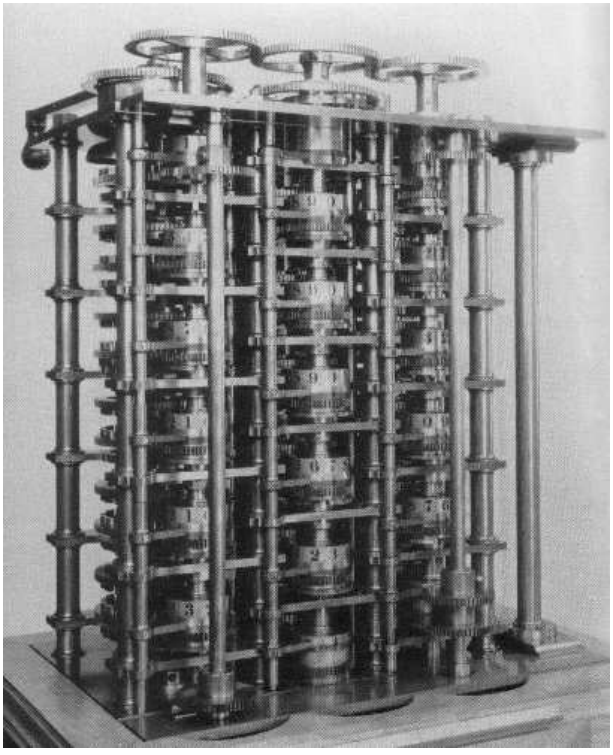
In 1666 Sir Samuel Morland (1625-1695) invented a mechanical calculator that could add and subtract. This machine was designed for use with English currency but had no automatic carry mechanism. Auxiliary dials recorded numerical overflows and had to be re-entered as addends.[vi] In 1673, Gottfried Wilhelm von Leibniz (1646 - 1716) designed a machine called the "Stepped Reckoner" that could mechanically perform all four mathematical operations using a stepped cylinder gear, though the initial design gave some wrong answers. This machine was never mass-produced because the high level of precision needed to manufacture it was not yet available.[vii] In 1774 Philipp-Matthaus Hahn (1739 - 1790) constructed and sold a small number of mechanical calculators with twelve digits of precision.

The advent of the Industrial Revolution, just prior to the start of the nineteenth century, ushered in a massive increase in commercial activity. This created a great need for automatic and reliable calculation. Charles Xavier Thomas (1791 - 1871) of Colmar, France invented the first mass-produced calculating machine, called the Arithmometer (left) in 1820. His machine used Leibniz's stepped cylinder as a digital-value actuator. However, Thomas' automatic carry system worked in every possible case and was much



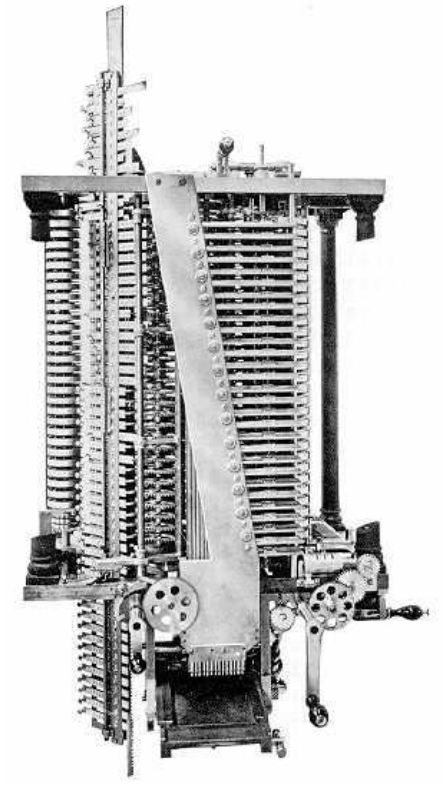
more robust than any predecessor. This machine was improved and produced for decades. Other models, designed by competitors, eventually entered the marketplace.

In 1786, J. H. Mueller, of the Hessian army, conceived the "Difference Engine" but could not raise the funds necessary for its construction. This was a special purpose calculating device that, given the differences between certain values where the polynomial is uniquely specified, can tabulate the polynomial values. This calculator would be useful for functions that can be approximated polynomially over certain intervals. The realization of the Difference Engine's mechanical computer prototype design would not occur until 1822, when conceived by Charles Babbage (1792 - 1871). In 1832, Babbage and Joseph Clement built a scaled-down prototype that could perform operations on 6-digit numbers and 2nd order or quadratic polynomials. A full-sized machine would be as big as a room and able to perform operations on 20-digit numbers and 6th order polynomials. Babbage's Difference Engine project was eventually canceled due to cost overruns. In 1843, George Scheutz and his son Edvard Scheutz, of Stockholm, produced a 3rd order engine with the ability to print its results. From 1989-91, a team at London's Science Museum built a fully functional Difference Engine based on Babbage's latest (1837), improved and simpler design, using modern construction materials and techniques. The machine could successfully operate on 31-digit numbers and 7th order differences.



The Difference Engine uses Sir Isaac Newton's method of differences. It works as follows: Consider the polynomial $p(x) = x^2 + 2x + 1$ and tabulate the values for p(0), p(0.1) , p(0.2) , p(0.3) , p(0.4). The table below contains the results of the polynomial values in the first column, the differences of each consecutive set of polynomial results in the second column, and the differences of each consecutive set of differences from the second column in the third column. For a 2^{nd} order polynomial, the third column will always contain the same value. Likewise, for an n^{th} order polynomial, column n+1 will always have the same value. To find p(0.5), start from the right column with value 0.02 and subtract this from the second column to get -0.29. Then subtract this value from the first column to get 2.25, which is the solution to p(0.5). This can be continued incrementally for greater p(x), indefinitely, by updating the table and repeating the algorithm.

p(0) = 1		
	1 – 1.21 = -0.21	
p(0.1) = 1.21		-0.21 – (-0.23) = 0.02
	1.21 – 1.44 = -0.23	
p(0.2) = 1.44		-0.23 – (-0.25) = 0.02
	1.44 – 1.69 = -0.25	
p(0.3) = 1.69		-0.25 – (-0.27) = 0.02
	1.69 – 1.96 = -0.27	
p(0.4) = 1.96		



This device impresses a zinc block, which prints the results of calculations on paper. This could be considered the first standalone computer printer.

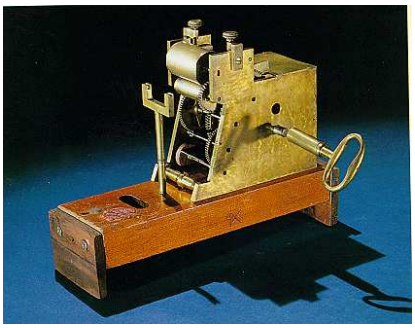
Babbage also invented the Analytical Engine, which was the first computing device designed to use read-only memory, in the form of punched cards, to store programs. This general-purpose mathematical device was very similar to electronic processes used in early computers. Later designs of this machine would perform operations on 40-digit numbers. The machine had a processing unit called the "mill" that contained two main accumulators and some special purpose auxiliary accumulators. It also had memory area called the "store", which could hold approximately 100 more numbers. To accept data and program instructions, the Analytical Engine would be equipped with several punch card readers in which the cards were linked together to allow forward and reverse reading. These linked cards were first used in 1801 by Joseph-Marie Jacquard to control the weaving patterns of a loom. The machine could perform conditional

branching called “jumps”, which allowed it to skip to a desired instruction. The device was capable of using a form of microcoding by using the position of studs on a metal barrel called the “control barrel” to interpret instructions. This machine could calculate an addition or subtraction operation in about three seconds, and a multiplication or division operation in about three minutes.

In 1843, Augusta Ada Byron (1815 - 1852), Lady Lovelace, mathematician, scientist and daughter of the famed poet Lord Byron, translated an article from French about Babbage’s Analytical Engine, adding her own notes. Ada composed a plan for the calculation of Bernoulli numbers, which is considered to be the first ever “computer program.” Though because it was never built, the algorithm was never run on Analytical Engine. In 1979, the U.S. Department of Defense honored the world’s first “computer programmer” by naming its own software development language as “Ada.”[viii]

George Boole (1815 - 1864) (right) wrote, "An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities" in 1854. This article detailed Boole’s new binary approach, which processed only two objects at a time (in a yes-no, true-false, on-off, zero-one type manner), to logic by incorporating it into mathematics and reducing it to a simple algebra, which presented an analogy between symbols that represent logical forms and algebraic symbols. Three primary operations were defined based on those in Set Theory: AND—intersection, OR—union, and NOT—compliment. This system was the beginning of the Boolean algebra that is the basis for many applications in modern electronic circuits and computation.[ix] Though his idea was either ignored or criticized by many of his peers, twelve years later, an American, Charles Sanders Peirce, described it to the American Academy of Arts and Sciences. He spent the next twenty years expanding and modifying the idea, eventually designing a basic electrical logic-circuit.

Processing and storage were not the only advancements made prior to the 20th century. There were also great improvements in communications technology. Samuel

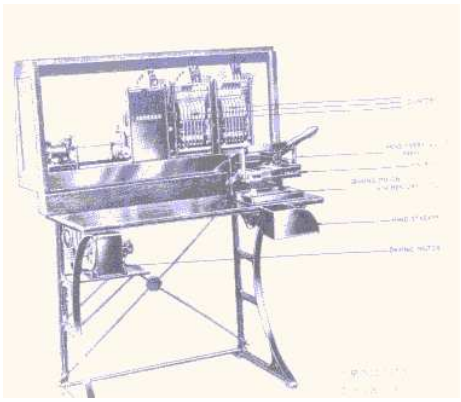


Morse (1791 -1872) conceived the telegraph in 1832 and had built a working model by 1835. This was the first device to communicate through the use of electricity. The telegraph worked by tapping out a message from a sending device (right) in Morse code, which was a series of dots-and-dashes that represented letters, numbers, punctuation and other symbols. These dots-and-dashes were converted into electrical impulses and sent, on the wire, to a receiver (left). The receiver converted the electrical impulses to an audible sound that represented the original dots-and-dashes. In 1844, he sent a signal from Washington to Baltimore over this communication device. By 1854 there was 23,000 miles of telegraph wire being used within the United States. This provided a much more efficient form of communication that greatly affected national socio-economic development.[x] In 1858, a telegraph cable was run across the Atlantic Ocean, providing communication service between the U.S. and England for less than a month. By 1861 a transcontinental cable connected the East and West coasts of the U.S. and by 1880, 100,000 miles of undersea cable had been laid.

The next great advancement in communication was Alexander Graham Bell's (1847 - 1922) invention of the "electrical speech machine" or telephone in 1876. This invention was developed from improvements that Bell made to the telegraph, which allowed more than one signal to be transmitted over a single set of telegraph wires, simultaneously. Within two years, he had set up the first telephone exchange in New Haven, Connecticut. He had established long distance connections between Boston, Massachusetts and New York City by 1884. The telecommunication industry would eventually reach almost every locality in the country, then the world. Bell's original venture evolved into larger companies and in 1881 American Bell Telephone Co. Inc. purchased Western Electric Manufacturing Company to manufacture equipment for Bell. In 1885, American Telephone and Telegraph Company (AT&T) were formed to extend Bell system long lines across the U.S. and in 1899 AT&T became the parent company of Bell, assuming

all assets. The Western Electric Engineering Dept. was organized in 1907 and a research branch to do scientific research and development was organized in 1911. On December 27, 1925, Bell Telephone Laboratories was created to consolidate the research labs from AT&T and Western Electric, which remained a wholly owned subsidiary of AT&T after the divestiture of the seven regional Bell companies. Bell Laboratories would eventually become one of the world's premier communication and computer research centers. One of Bell Labs contributions to computing was the development of UNIX by Dennis Ritchie and Kenneth Thomson in 1970. In 1991, AT&T acquired NCR, formerly National Cash Register, which became AT&T Global Information Solutions.[xi]

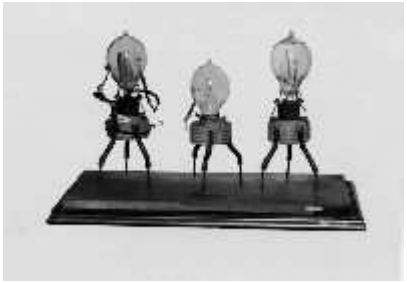
The explosion in population growth between 1880 and 1890, due to increased birth rates and immigration, created a great dilemma for the Census Bureau. During this time, Herman Hollerith (right) was a statistician for the Census Bureau and was responsible to solve problems related to the processing of large amounts of data from the 1880 US census. He was attempting to find ways of manipulating data mechanically as was suggested to him by Dr. John Shaw Billings. In 1882, Hollerith joined MIT to teach mechanical engineering and also started to experiment with Billings' suggestion by studying the operation of the Jacquard loom. Though he found that the loom's operation was not useful for processing data, he determined that the punched cards were very useful for storing data. In 1884, Hollerith devised a method to convert the data stored on the punched cards into electrical impulses using card-reading device. He also developed a typewriter-like device to record the data on the punched cards, which changed very little in its design over the next 50 years. The card readers used pins that pass through the holes in the cards creating electrical contacts, where the impulses from these contacts would activate mechanical counters to manipulate and tally the data. This system was successfully demonstrated in 1887 by tabulating mortality statistics and won the bid to be used to tabulate the 1890 Census data.



Hollerith had Pratt and Whitney manufacture the punching devices and the Western Electric Company to manufacture the counting devices. The Census Bureau's new system was ready by 1890 and processing the first data by September the same year. The count was completed by December 12, 1890 revealing that the total population of the United States to be 62,622,250. The count was not only completed eight times faster than if it was performed manually, it also allowed the gathering

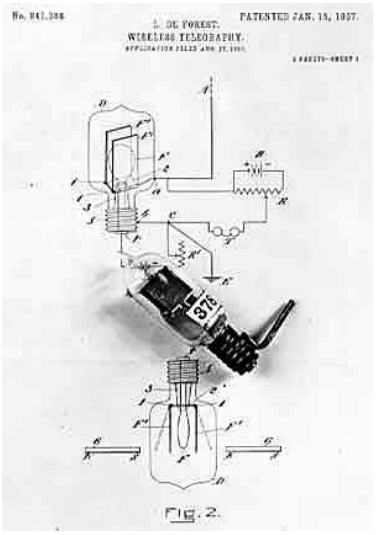
of more data than was possible before about the country's population, such as number of children in family, etc. Hollerith founded the Tabulating Machine Company in 1896 to produce his improved counting machines and other inventions, one of which automatically fed the cards into the counting machines. His system was used again for the 1900 Census but because Hollerith demanded more that the cost to count the data by hand, the Census Bureau was forced to develop its own system. In 1911, Hollerith's company merged with another company, becoming the Computer Tabulating Recording Company but was nearly forced out of the counting machine market due to fierce competition from new entrants. Hollerith retired his position of consulting engineer in 1921. Because of the efforts Thomas J Watson, who joined the company in 1918, the company reestablished its position as a leader in the market by 1920. In 1924, Computer Tabulating Recording Company was renamed as International Business Machines Corporation (IBM). By 1928, punch card equipment will be attached to computers as output devices and will also be used by L. J. Comrie to calculate the motion of the moon.[xii]

In 1895, Italian physicist and inventor Guglielmo Marconi sent the first wireless message. Prior to his first transmission, Marconi studied the works of Heinrich Hertz (1857-1894) and later started to experiment with Hertzian waves to transmit and receive messages over increasing distances without the use of wires. The messages were sent in Morse code. He patented his invention in 1896. After years of experimentation and improvement, especially with respect to distance, in 1897 Marconi named his company as the Wireless Telegraph and Signal Company. After a series of takeovers and mergers, this company eventually became part of the General Electric Company (GEC), which was eventually renamed Marconi Corporation plc in 2003.[xiii]



In 1904, radio technology was improved by the invention of the two-electrode radio rectifier, which was the first electron tube, also called the oscillation valve or thermionic valve (left). It is credited to John Ambrose Fleming, a consultant to the Marconi Company. This device was much more sensitive to radio signals then its predecessor, the coherer. This invention inspired all

subsequent developments in wireless transmission. In 1906, Lee de Forest improved the thermionic valve by adding a third electrode and a grid to control and amplify signals, creating a new device called an Audion. This device was used to detect radio waves and convert the radio frequency (RF) to an audio frequency (AF), which could be amplified through a loudspeaker or headphones. By 1907 gramophone music was regularly broadcast from New York over radio waves.[xiv]

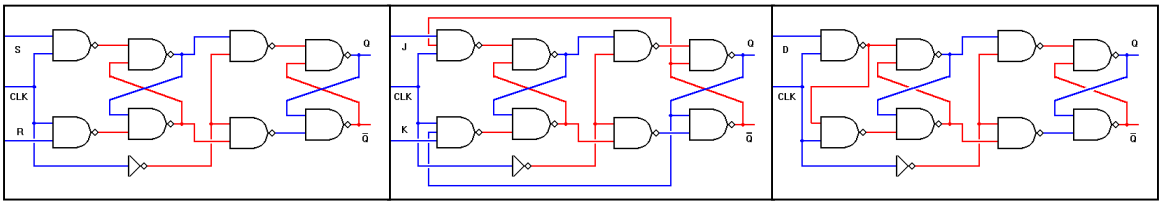


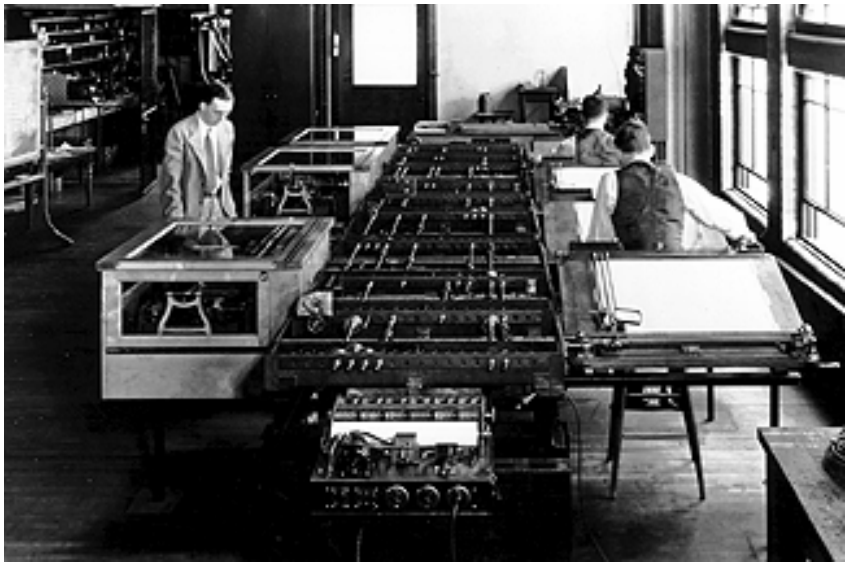
In 1907, both A. A. Campbell-Swinton ()(left) and Boris Rosing () independently suggest using cathode ray tubes to transmit images. Though intended for television, the cathode ray tube has made a valuable contribution to computing by providing a human readable interface with computational devices. In a letter to Nature magazine, Swinton describes first full description of an all-electronic television system as:

"Distant electric vision can probably be solved by the employment of two beams of kathode rays (one at the transmitting and one at the receiving station) synchronously deflected by the varying fields of two electromagnets placed at right angles to one another and energised by two alternating electric currents of widely different frequencies, so that the moving extremities of the two beams are caused to sweep synchronously over the whole of the required surfaces within the one-tenth of a second necessary to take advantage of visual persistence. Indeed, so far as the receiving apparatus is concerned, the moving kathode beam has only to be arranged to impinge on a suitably sensitive fluorescent screen, and given suitable variations in its intensity, to obtain the desired result."

In 1927, during a television demonstration, Herbert Hoover's face is the first image broadcast in the U.S., using telephone wires for the voice transmission. Vladimir Zworykin invented the cathode ray tube (CRT) in 1928. It eventually became the first computer storage device. Color television signals were successfully transmitted in 1929 and first broadcast in 1940.

In 1911, while studying the effects of extremely cold temperatures on metals such as mercury and lead, physicist Heike Kamerlingh Onnes discovered that they lost all resistance at certain low temperatures just above absolute zero. This phenomenon is known as superconductivity. In 1915, another physicist, Manson Benedicks, discovered that alternating current could be converted to direct current by using a germanium crystal, which eventually leads to the use of microchips. In 1919, U.S. physicists William Henry Eccles (1875 - 1966) and F.W. Jordan () invented the flip-flop, the first electronic switching electric circuit, which was critical to high-speed electronic counting systems. The flip-flop is a digital logic hardware circuit that can switch or toggle between two states controlled by its inputs, which is similar to a one-bit memory. The three common types of flip-flop are: the SR flip-flop, the JK flip-flop and the D-type flip-flop (shown below).



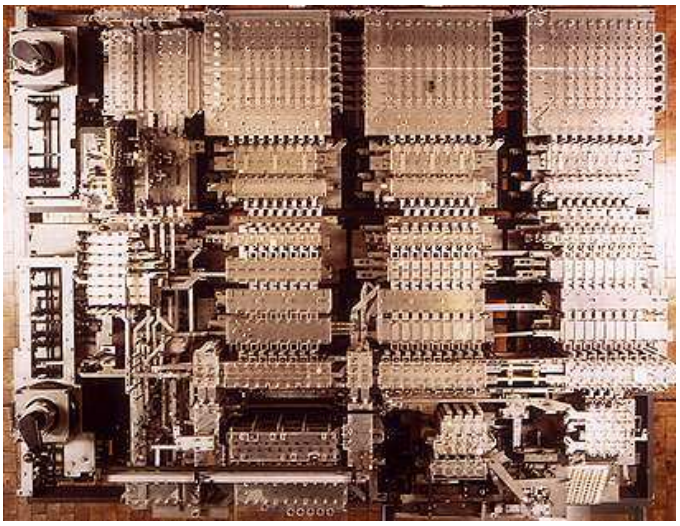


In 1925, Vannevar Bush (1890 - 1974) developed the first analog computer to solve differential equations. These analog computers were mechanical devices that used large gears and other mechanical parts to solve equations. The first working machine was completed in 1931 (left). In 1945, he published an article in the Atlantic Monthly called, "As We May Think, which described a theoretical device called a memex. This device uses a microfilm search system, which is very similar to hypertext, using a concept that he called associative trails. His description of the system is:

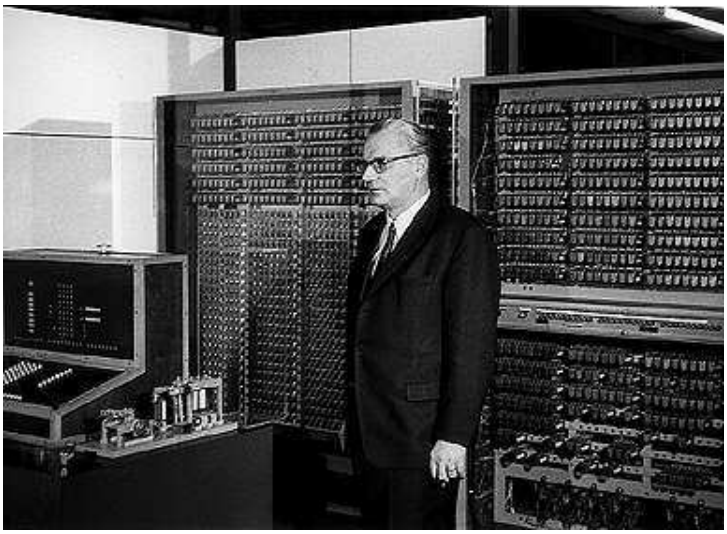
"The owner of the memex let us say, is interested in the origin and properties of the bow and arrow. Specifically he is studying why the short Turkish bow was apparently superior to the English long bow in the skirmishes of the Crusades. He has dozens of possibly pertinent books and articles in his memex. First he runs through an encyclopedia, finds an interesting but sketchy article, leaves it projected. Next, in a history, he finds another pertinent item, and ties the two together. Thus he goes, building a trail of many items. Occasionally he inserts a comment of his own, either linking it into the main trail or joining it by a side trail to a particular item. When it becomes evident that the elastic properties of available materials had a great deal to do with the bow, he branches off on a side trail which takes him through textbooks on elasticity and physical constants. He inserts a page of longhand analysis of his own. Thus he builds a trail of his interest through the maze of materials available to him."

In 1934, Konrad Zuse (1910 - 1995) was an engineer working for Henschel Aircraft Company, studying stresses caused by vibrations in aircraft wings. His work involved a great deal of mathematical calculation. To aid him in these calculations, he developed ideas on how machines should perform calculations. He determined that these machines should be freely programmable by reading a sequence of instructions from a punched tape and that the machine should make use of both the binary number system and a binary logic system to be capable of using binary switching elements. He designed a semi-logarithmic floating-point unit representation, using an exponent and a mantissa, to calculate both very small and very large numbers. He developed a "high performance adder", which included a one-step carry-ahead and precision arithmetic exceptions handling. He also developed an addressable memory that could store arbitrary data. He devised a control unit to control all other devices within the machine along with input and output devices that convert numbers from binary to decimal and vice versa.

By 1936 he completed the design for the Z1 computer (top next page), which he constructed in his parents' living room by 1938. This was a completely mechanical unit



based on his previous design. Though unreliable, it had the ability to store 64 words, each 22 bits in length (8 bits for the exponent and sign, and 14 bits for the mantissa), in its memory, which consisted of layers of metal bars between layers of glass. Its arithmetic unit was constructed from a large number of mechanical switches and had two 22-bit registers. The machine was freely programmable with the use of a punched tape. The device also had the prescribed control unit and addressable memory, making it the world's first programmable binary computing machine, with a clock speed of 1-Hertz. The picture above is a topside view of the Z1, which is very similar in appearance to a silicon chip. At first the machine was not very reliable. However, it functioned reliably by 1939.

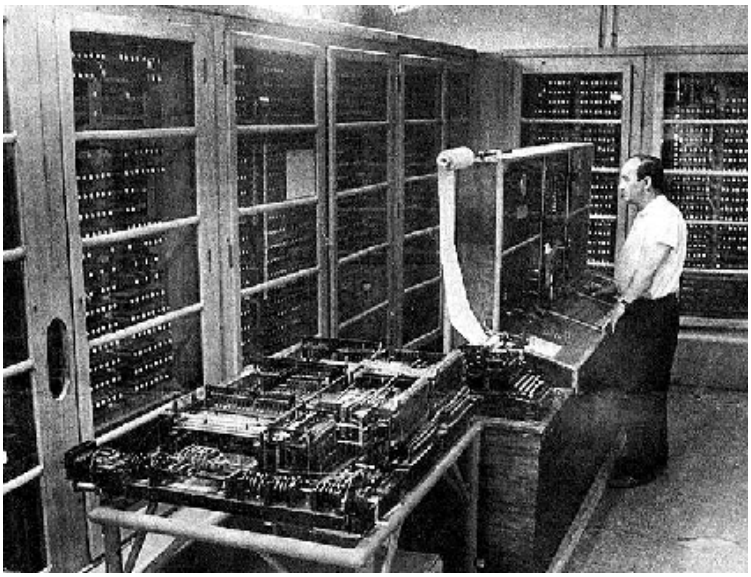


The Z2 was an experimental machine similar to the Z1 but used 800 relays for the arithmetic unit instead of mechanical switches. This machine proved that relays were reliable, which prompted Zuse to design and build the Z3 using relays. The Z3 was constructed between 1938 and 1941 in Berlin. The Z3 used relays for the entire machine and had a 64-word memory, consisting of 22-bit floating-point numbers. The Z3 was the first reliable, fully functional, freely programmable computer based on the binary floating-point number and a switching system, which had the capability to perform complex arithmetic calculations. It had a clock speed of 5.33 Hertz and could perform a multiplication operation in 3 seconds. This machine contained all the components except the ability to store the program in the memory together with the data that was described by the von Neumann et al machine in 1946. In 1998, Raul Rojas proved that the Z3 was

a truly universal computer in the sense of a Turing machine. The picture above is Zuse along with his 1961 reconstruction of the Z3. Allied bombing, during World War II, destroyed the original Z3.

An example program from "The Life and Work of Konrad Zuse" Web Site, authored by Horst Zuse, listed in the references section, for the Z3 is the calculation of the polynomial: ((a4x + a3)x + a2)x + a1, where a4, a3, a2, and a1 would first be loaded into the memory cells 4, 3, 2, and 1.

```
Lu     To call the input device for the variable x
Ps 5   To store variable x in memory word 5
Pr 4   Load a4 in Register R1
Pr 5   Load x in Register R2
Lm     Multiply: R1 := R1 x R2
Pr 3   Load a3 in Register R2
Ls1    Add: R1 := R1 + R2
Pr 5   Load x in R2
Lm     Multiply: R1 := R1 x R2
Pr 2   Load a2 in Register R2
Ls1    Add: R1 := R1 + R2
Pr 5   Load x in Register R2
Lm     Multiply: R1 := R1 x R2
Ppr 1  Load a1 in Register R2
Ls1    Add: R1 := R1 + R2
Ld     Shows the result as a decimal number
```



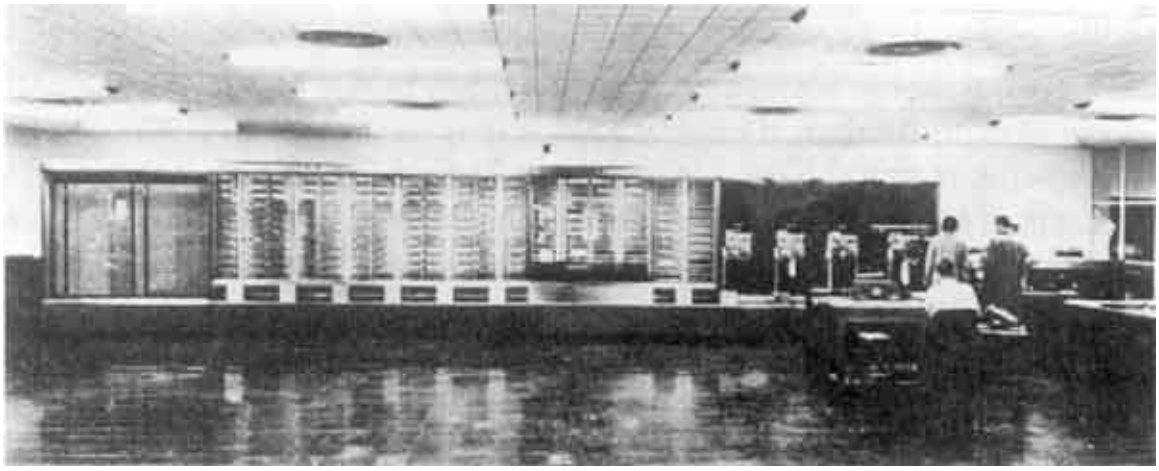
The program above is very similar to the assembly code that is used in modern computers. From 1942 to 1946 Zuse began to develop ways to program computers. To aid engineers and scientists in the solution of complex problems, he developed the Plankakül (plan calculus) programming language. This precursor to today's algorithm-type languages was the world's first programming language and was intended for a logical machine. A logical machine could do more than just numerical calculations, of which the algebraic machines (Z1, Z2, Z3 & Z4) that he had previously designed are limited. The picture on the left is the Z4 model, completed in 1945 and reconstructed in

1950, which used a mechanical memory, similar to that in the Z1, and had 32-bit words. By 1955, this machine had the added abilities to call subprograms, through a secondary punch tape reader, and use a conditional branch instruction.

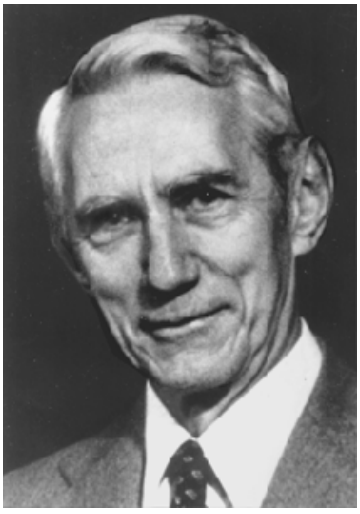
In 1942, Zuse built the S1, a special purpose computer to measure the wing surface area of airplanes, with 600 relays and 12-bit binary words. This machine was destroyed in 1944. Zuse improved this model with the construction of the S2. This machine used approximately 100 clock gauges to automatically scan the surface of wings. This computer was most likely the first machine to use the concept of a process. It was destroyed in 1945. In 1949, he founded Zuse KG, Germany's first computer company. In 1952, Zuse KG constructed the Z5 for optical calculations, an improved version of the Z4, which was about six times faster. It had many punch card readers for data and program input, a punch card writer to output data and could handle 32-bit floating-point numbers. In 1957, Zuse KG constructed the Z22 that contained an 8192-word magnetic drum and was the first stored program computer. In 1961, Zuse KG built the Z23, which was based on the same logic as and three times faster than the Z22, and was the first transistor-based computer. In 1965, his company produced the Z43, which was the first modern transistor computer to use TTL logic. The TTL (transistor-transistor-logic) type digital integrated circuit (IC) uses transistor switches for logical operations. In 1956, Siemens AG purchased Zuse KG.[xv]

In 1937, Howard Aiken (1900 - 1973) proposed a machine that could perform four fundamental operations of arithmetic, addition, subtraction, multiplication and division, in a predetermined order to Harvard University, which was forwarded to IBM. His research had led to a system of differential equations that could only be solved using a prohibitive amount of calculations using numerical techniques and which had no exact solutions. His report stated:

"... whereas accounting machines handle only positive numbers, scientific machines must be able to handle negative ones as well; that scientific machines must be able to handle such functions as logarithms, sines, cosines and a whole lot of other functions; the computer would be most useful for scientists if, once it was set in motion, it would work through the problem frequently for numerous numerical values without intervention until the calculation was finished; and that the machine should compute lines instead of columns, which is more in keeping with the sequence of mathematical events."



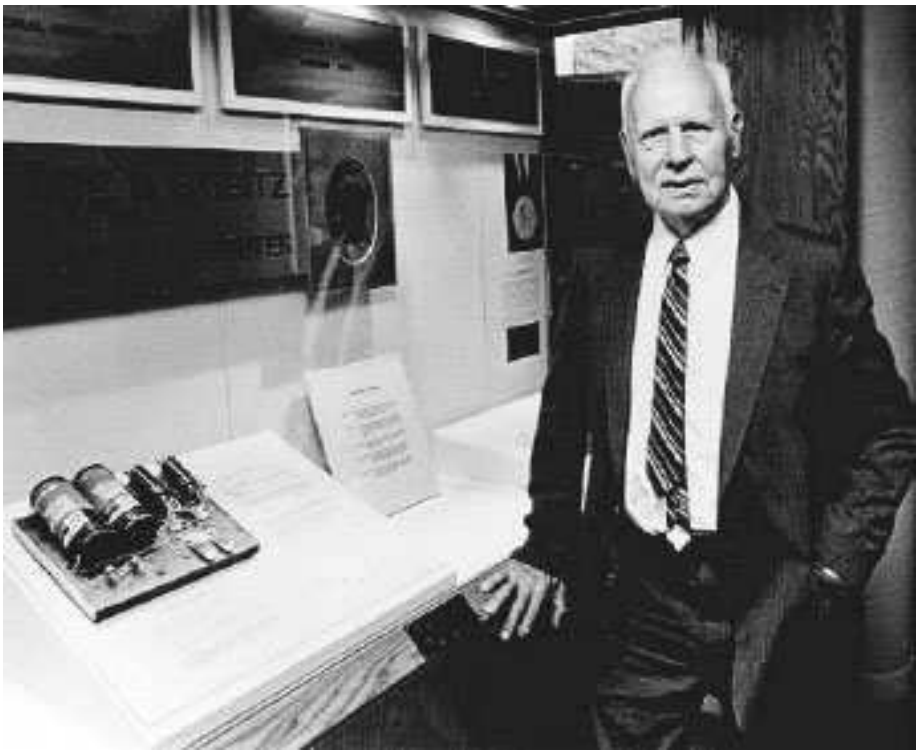
Aiken, working with IBM engineers, developed the ASCC computer (Automatic Sequence Controlled Calculator), which was capable of five operations, addition, subtraction, multiplication, division and reference to previous results. Though it ran on electricity and the major components were magnetically operated switches, this machine had a lot in common with Babbage's analytical engine. Construction of the machine started in 1939 at the IBM laboratories, Endicott and was completed in 1943. The machine weighed 35 tons, had more than 500 miles of wire, and used vacuum tubes and relays to operate. The machine had 72 storage registers and could perform operations to 23 significant figures. The machine instructions were entered on punched paper tapes, and punched cards were used to enter input data. The output was either in the form of punched cards or printed by means of an electric typewriter. The machine was moved to Harvard University, where it was renamed the Harvard Mark I, pictured above. The US navy used this machine in the Bureau of Ordnance's Computation Project for gunnery and ballistics calculations, which was performed at Harvard. In 1947, Aiken completed the Harvard Mark II, which was a completely electronic computer. He also worked on the Mark III (the first computer to contain a drum memory) and Mark IV computers, and made contributions in electronics and switching theory.[xvi]



In 1937, Claude Shannon (1916 - 2001) wrote his Master's thesis, "A Symbolic Analysis of Relay and Switching Circuits", using symbolic logic and Boole's algebra to analyze and optimize relay-switching and computer circuits. It was published in A.I.E.E. Transactions in 1938. For this work, Shannon was awarded the Alfred Nobel Prize of the combined engineering societies of the United States in 1940. In 1948, Shannon published his most important work on information theory and communication, "A

Mathematical Theory of Communication", where he demonstrated that all information sources have a "source rate" and all communication channels have a "capacity", both measurable in bits-per-second, and that the information can be transmitted over the channel if and only if the capacity of the channel is not exceeded by the source rate. He also published works related to cryptography and the reliability of relay circuits, both with respect to transmission in noisy channels.[xvii]

George Stibitz, a Bell Labs researcher, created the first electromechanical circuit that could control binary addition from old relays, batteries, flashlight bulbs, wires and tin strips in 1937. He realized that Boolean logic could be used for electromechanical telephone relays. He incorporated this binary adder (picture on left with Stibitz) prototype in his Model K digital calculator. Over the next two years, Stibitz and his associates at Bell Labs devised a machine to perform all four basic math operations on complex numbers. It was initially called the Complex Number Calculator but was renamed the Bell Labs Model Relay Computer (also known as the Bell Labs Model 1) in 1949. This machine is considered to be the world's first electronic digital computer. Its electromechanical brain consisted of 450 telephone relays and 10 crossbar switches, and three teletypewriters provided input to the machine. It could find the quotient of two eight-place complex numbers in about 30 seconds. Stibitz brought one of the typewriters to an American Mathematical Association meeting in 1940 at Dartmouth and performed the world's first demonstration of remote computing by using phone lines to communicate with the Complex Number Calculator, which was in New York.[xviii]

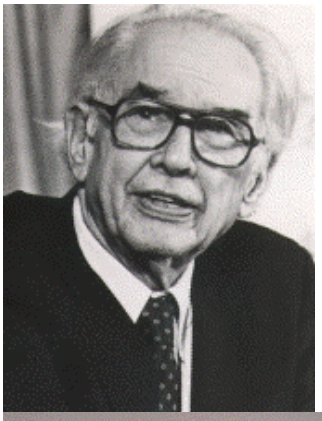


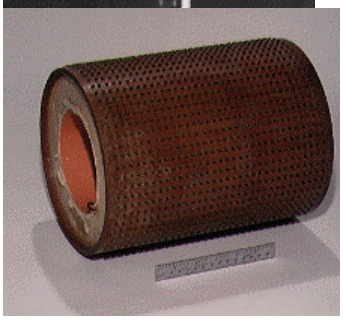
In 1937, Alan Turing (1912 - 1954) published his paper “On Computable Numbers, with an application to the Entscheidungsproblem (decision problem)”. In this paper, he introduced the Turing Machine, which was an abstract machine capable of reading or writing symbols and moving between states, dependent upon the symbol read from a bi-directional, movable tape, using a set of finite rules listed in a finite table. This machine demonstrated that every method found for describing ‘well-defined procedures’, introduced by other mathematicians, could be reproduced on a Turing machine. This statement is known as the Church-Turing thesis and is a founding work of modern computer science, which defined computation and its absolute limitation. His definition of computable was that a problem is ‘Calculable by finite means’. In 1938, his Ph.D. thesis, which was published as “Systems of Logic based on Ordinals” in 1939, Turing addressed uncomputable problems.

During World War II, Turing worked at Bletchley Park, the British government's wartime communications headquarters. His main task was to master the Enigma (pictured right), the German enciphering machine, which he was able to crack, providing the Allies with valuable intelligence. His contributions made him a chief scientific figure in the fields of computation and cryptography. After the war, he was interested in the comparison of the power of computation and the power of the human brain. He proposed the possibility that a computer, if properly programmed, could rival the human mind. In 1950, Turing wrote his famous paper "Computing Machinery and Intelligence," which, along with his previous work, founded the study of ‘Artificial Intelligence’. This paper introduces ‘the imitation game’, which is a test to determine if a computer program has intelligence. This game is now referred to as the Turing Test. Turing describes the original imitation game as:

"The new form of the problem can be described in terms of a game which we call the 'imitation game.' It is played with three people, a man (A), a woman (B), and an interrogator (C) who may be of either sex. The interrogator stays in a room apart from the other two. The object of the game for the interrogator is to determine which of the other two is the man and which is the woman. He knows them by labels X and Y, and at the end of the game he says either "X is A and Y is B" or "X is B and Y is A." The interrogator is allowed to put questions to A and B."

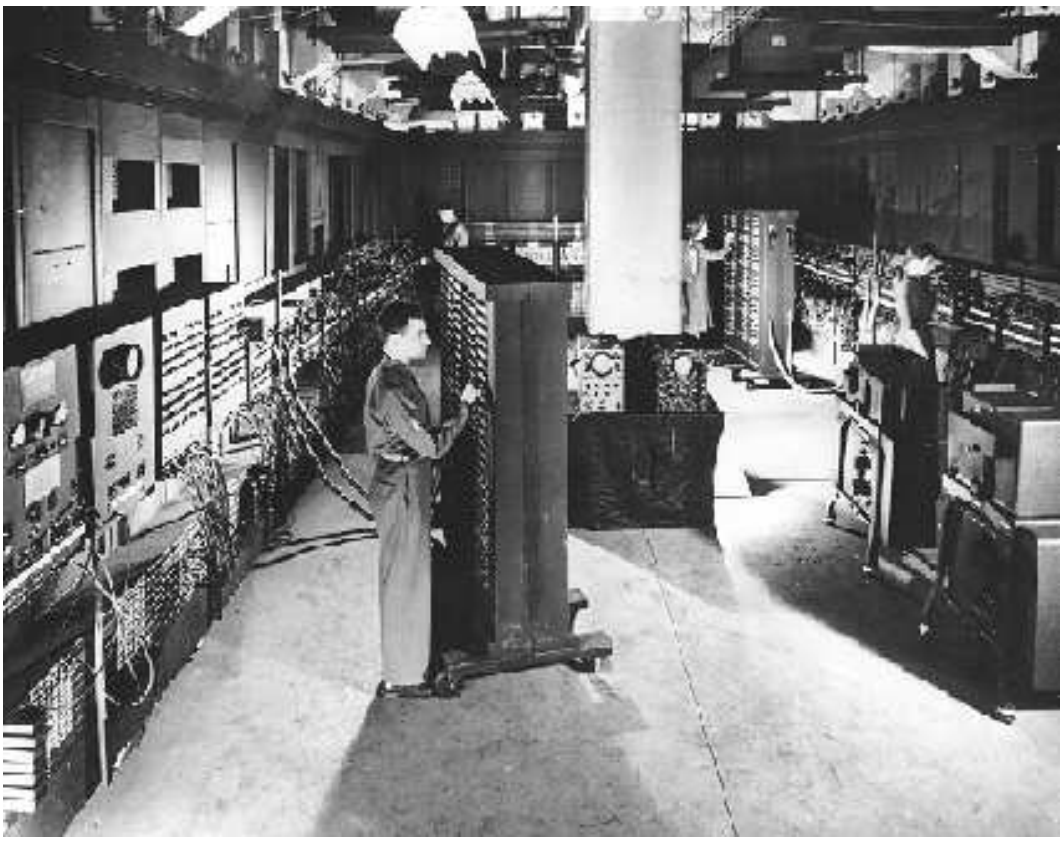
The idea in the Turing Test is that the interrogator (C) is actually communicating with human (A), a machine (B). The interrogator asks the two candidates questions to decide their identities, as above with the man and woman. In order to prove that it's program is intelligent, the machine must fool the interrogator into choosing it as the human.[xix]





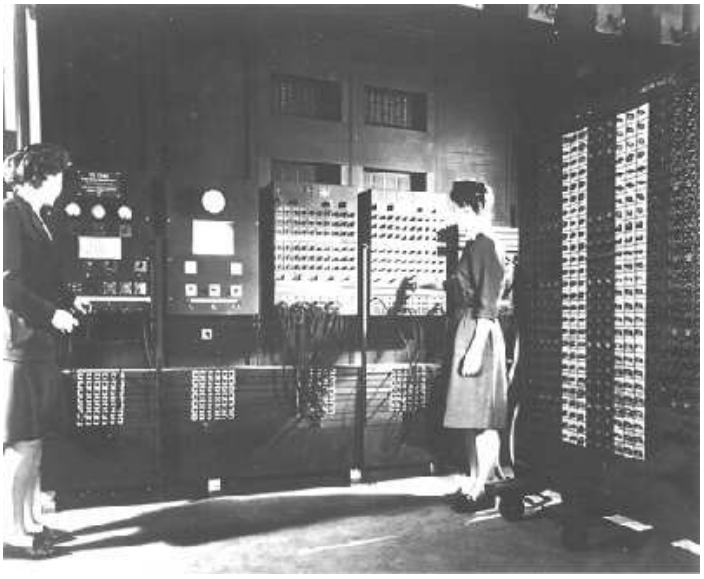
Between 1937 and 1938, John Vincent Atanasoff (far left) and Clifford Berry devised the principals for the ABC machine (right), an electronic-digital machine that would lead to advances in digital computing machines. This non-programmable binary machine's construction began in 1941 but was stopped in 1942 due to World War II before becoming operational. This machine employed capacitors to store electrical charge that could correspond to numbers in the form of logical 0's and 1's. This was the first machine to demonstrate electronic techniques in calculation and to use regenerative memory. It contained 300 vacuum tubes in its arithmetic unit and 300 more in its control unit. The capacitors were affixed inside of 12-inch tall by 8-inch diameter rotating Bakelite (a thermosetting plastic) cylinders (shown below) with metal contact bands on their outer surface. Each cylinder contained 1500 capacitors and could store 30 binary numbers, 50 bits in length, which could be read from or written to the metal bands of the rotating cylinder. The input data was loaded on punched cards. Intermediate data was also stored on punched cards by burning small spots onto the cards with electric sparks, which could be re-read by the computer at some

later time by detecting the difference in electrical resistance of the carbonized burned spots. This machine could also convert from binary to decimal and vice versa.[xx]



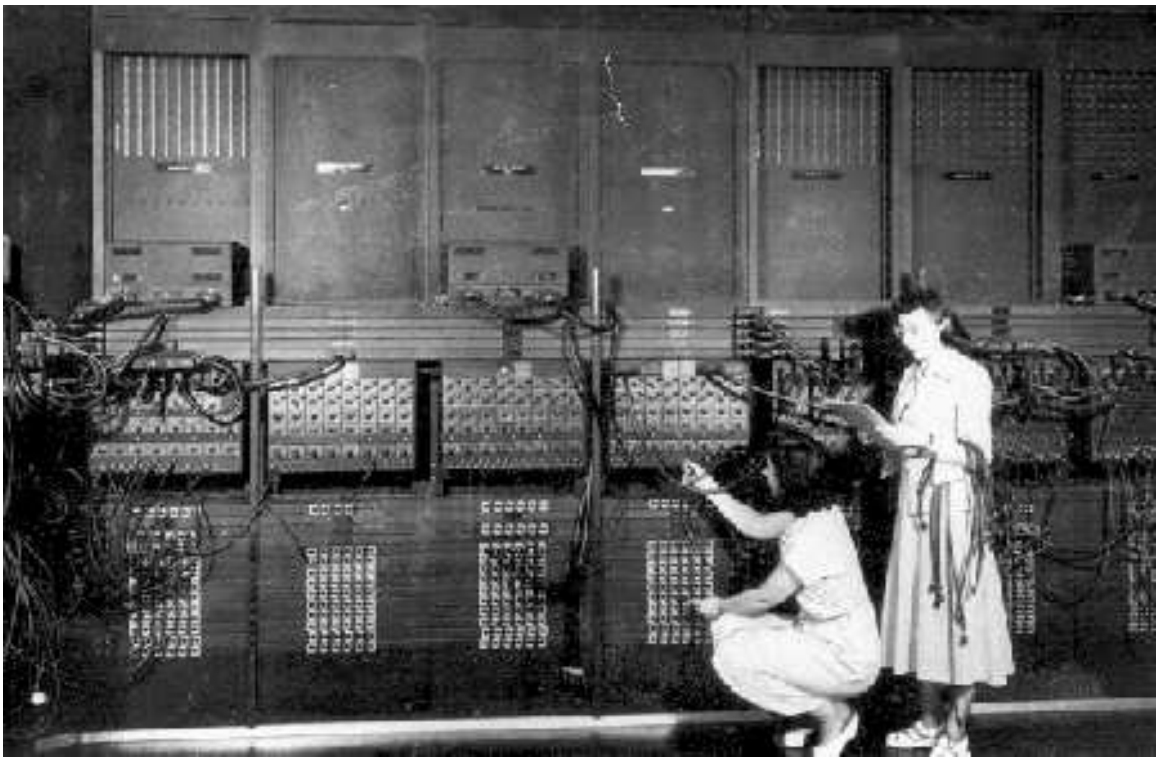
In 1943, the U.S. Army contracted with the Moore School of Electrical Engineering, University of Pennsylvania, for the production of the Electrical Numerical Integrator and Computer (ENIAC), which would be used to calculate ballistic tables, which was designed by J. Presper Eckert (1919-1995) and John Mauchly (1907-1980). The 30-ton machine with approximately 18,000 vacuum tubes was completed in 1946 and was contained in a 30' by 50' room.

The ENIAC was a general-purpose digital electronic computer that could call subroutines. It could reliably perform 5,000 additions or 360 multiplications per second, which was between 100 and 1000 times faster than existing technology. At the time of its introduction, ENIAC was the world's largest single electronic apparatus. This machine was separated into thirty autonomous units. Twenty of these were accumulators, which were ten-digit, high-speed adding machines with the ability to store results. These accumulators used electronic circuits called ring counters, a loop of bistable devices (flip-flops) interconnected in such a manner that only one of the devices may be in a specified

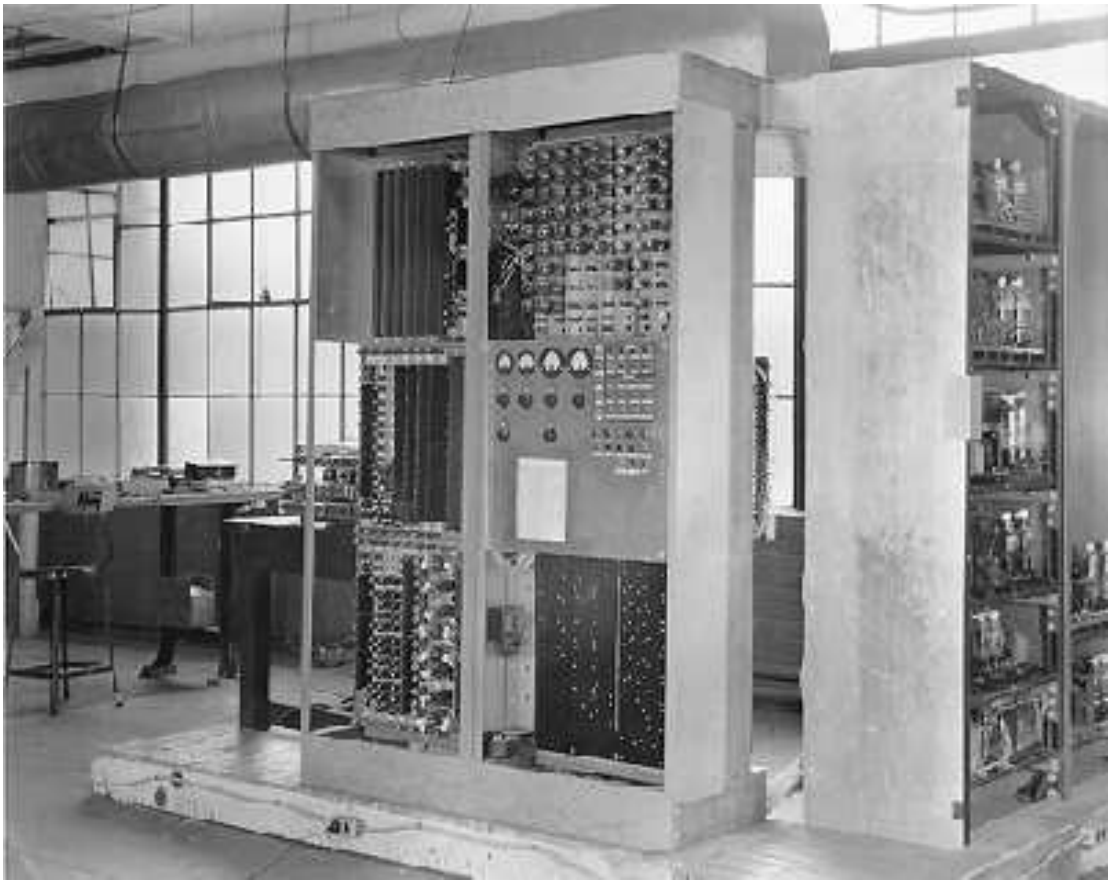


state at one time, to count each of its digits from 0 to 9 (a decimal arithmetic unit). The machine also had a multiplier and divider-square rooter, which special devices to accelerate their respective arithmetic operations.

A “computer program” on ENIAC was entered by using wires to connect different units of the machine as to perform operations is a required sequence. The picture on the left shows two women entering a program, which was a very difficult task. The machine was controlled by a sequence of electronic pulses, in which each unit on the machine could issue a pulse to cause one or more other units to perform a computation. The control and data signals on ENIAC were identical, typically were 2 microsecond pulses placed at ten microsecond intervals, which could allow for the output



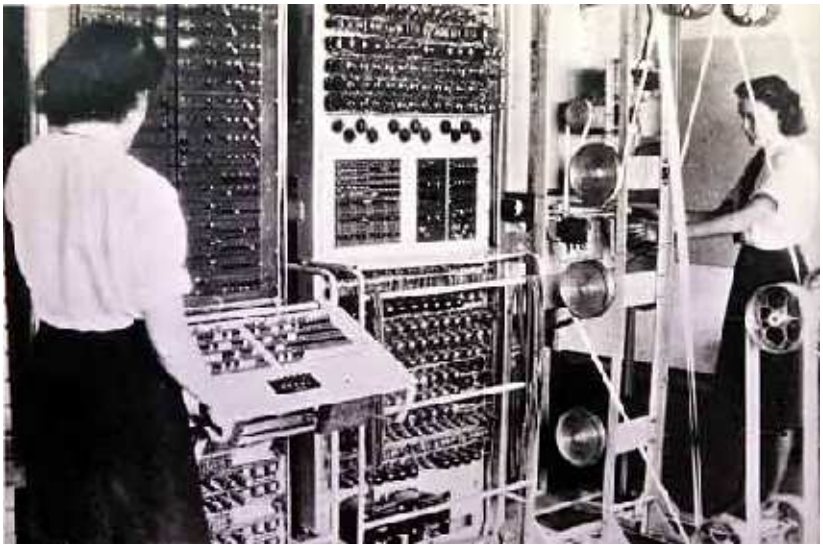
of an accumulator to be attached to the input of a control line of another accumulator. This could allow data-sensitive operations or operations based on data content. It also had a unit called the “Master Programmer”, which performed nested loops or iterations. ENIAC’s units could operate simultaneously, performing parallel calculations. Eventually this machine could perform IF-THEN conditional branches. It is likely that this was the first machine with this operation.[xxi]



In 1944, because of suggested improvements from people involved with the project, the U.S. Army extended the ENIAC project to include research on Electronic Discrete Variable Automatic Computer (EDVAC), a stored program computer. At about this time, John von Neumann (1903 - 1957) visited the Moore School to take part in discussions regarding EDVAC’s design. He is best known for producing the best-recognized formal description of a modern computer, based on a stored program computer, known as the von Neumann architecture, in his 1946 paper "First Draft of a report to the EDVAC". The basic elements of this architecture are:

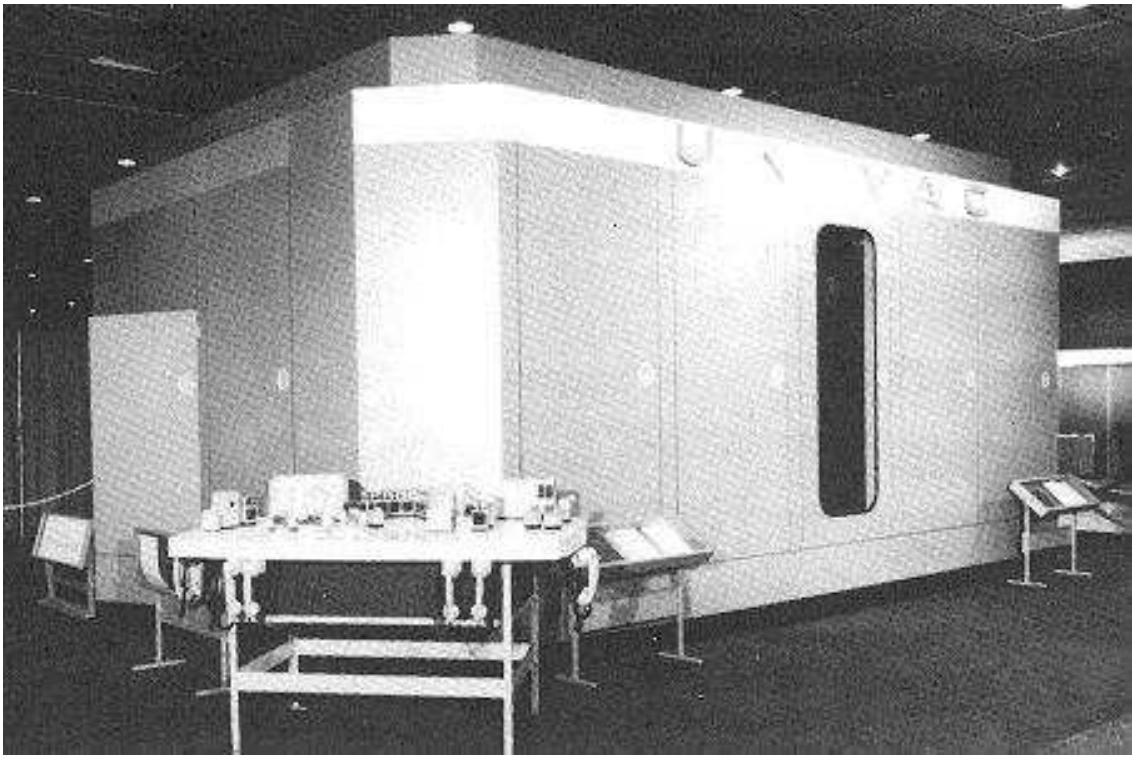
- A memory, which contains both data and instructions and also allows both data and instruction locations to be read from, and written to, in any order.
- A calculating unit, which can perform both arithmetic and logical operations on the data.
- A control unit, which can interpret retrieved memory instructions and select alternative courses of action based on the results of previous operations.

The EDVAC was a multipurpose binary computing machine with a memory capacity of 1,000 words, which was more than any other computing device of its time. Its memory worked by using mercury delay lines, tubes of mercury in which electrical impulses were bounced back and forth, creating a two-state device for storing 0's and 1's, which could be assigned or retrieved at will. It used 12 of 16 possible 4-bit instructions and each word in memory had 44 bits. The integer range was $\pm 1\text{-}2^{43}$ and the floating-point numbers had a 33-bit mantissa, 10 bit exponent and 1 bit for the sign, with a range $\pm (1\text{-}2^{-33})2^{511}$. It had approximately 10,000 crystal diodes and 4,000 vacuum tubes. Its average error-free up-time was about 8 hours. Its magnetic drum could hold 4,608 words 48 bits in length and a block transfer length of between 1 and 384 words. It also had a magnetic tape storage system that could store 112 characters per inch on a magnetic wire that was between 1,250 and 2500 feet long with a variable block length of between 2 and 1024 words also 48 bits long. During searches of the tape the machine could be released for computation and data read from the tape could be automatically re-recorded to the same place on the tape. EDVAC's input devices consisted of a photoelectric tape reader could read 78 words per second and an IBM card reader that could read 146 cards per minute at 8 words per card. The output devices were a 30 word per minute paper tape perforator, a 30 word per minute teletypewriter and a 1000 word per minute cardpunch. This machine had a clock speed of 1 MHz and was a significant improvement over ENIAC.[xxii]



Thomas Flowers and crew started construction on the Mark 1 COLOSSUS computer in 1943 at Dollis Hill Post Office Research Station in the U.K. Max Newman and associates of Bletchley Park ('Station X'), Buckinghamshire, designed this machine, which was primarily intended for

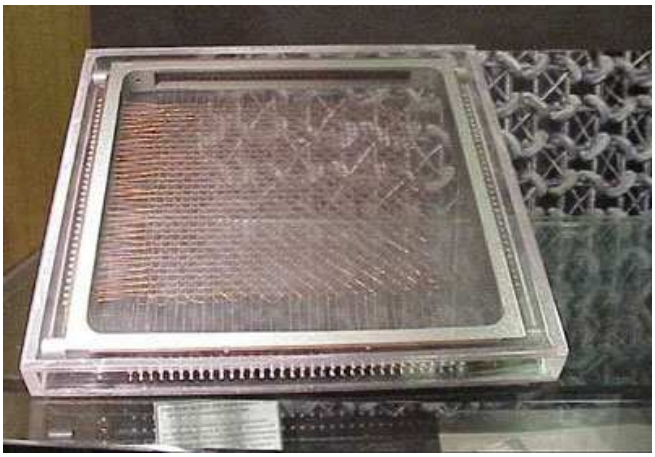
cryptanalysis of German Fish teleprinter ciphers used during World War II. This electromechanical attempt at a one-time pad was the German military's most secure method of communication. Prior to knowledge of Zuse's Z3, this was considered to be the first totally electronic computing device, using only vacuum tubes as opposed to relays in the Z3. This special purpose computer was equipped with very fast optical punch card readers for input. Nine of the improved Mark II machines were constructed and the original COLOSSUS Mark I was converted, for a total of ten machines. These machines were considered to be of the highest level of secrecy. After the end of the war, by direct orders from Churchill, all ten machines were destroyed—reduced into pieces no larger than a man's hand. The COLOSSUS, Heath Robinson (precursor to the COLOSSUS) and the Bombe (a machine designed by Alan Turing) are all in the process of reconstruction to preserve these important achievements.



The Universal Automatic Computer I (UNIVAC I) was designed by J. Presper Eckert and John Mauchly in 1947. The machine, constructed by Eckert-Mauchly Computer Corporation, founded by Eckert and Mauchly in 1946 but later purchased by Sperry-Rand, was delivered to the US Census Bureau in 1951 at a cost of $159,000. By 1953, three UNIVACs were in operation and by 1958 there were forty-six in the service of government departments and private organizations. Rand sold the later machines for more than $1,000,000 each.

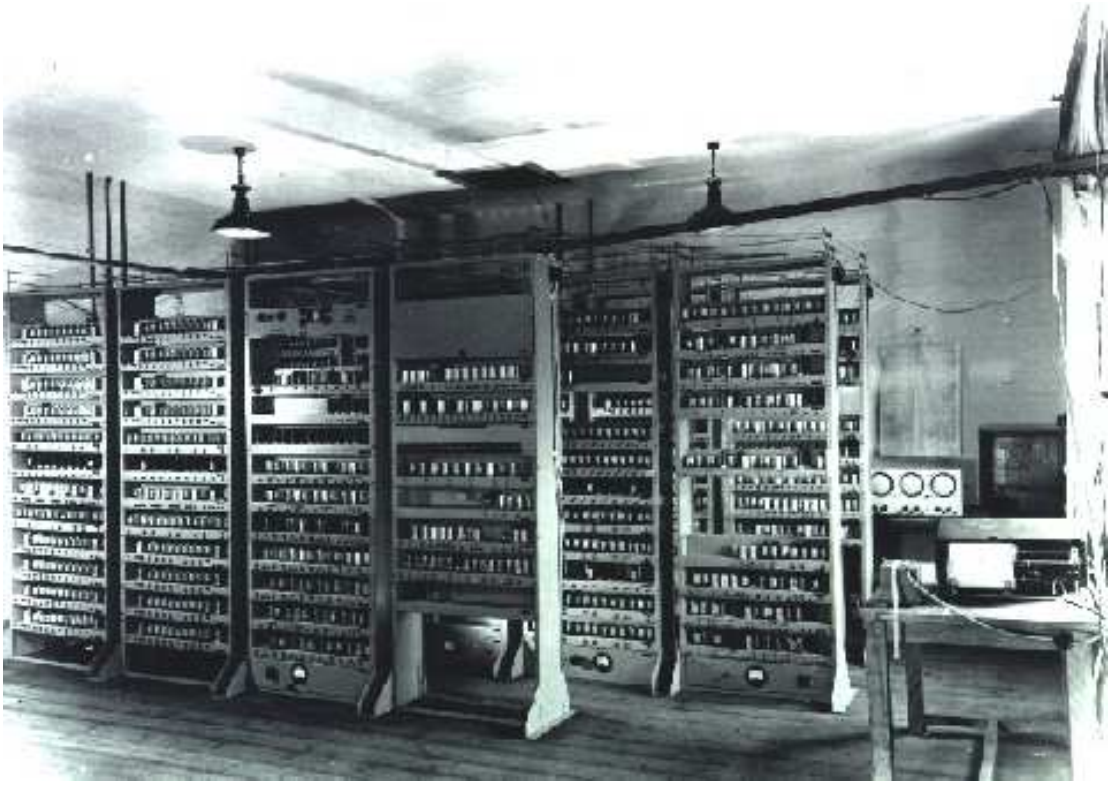
UNIVAC's input consisted of 12,800 character per second magnetic tape reader, a 240 card per minute card to tape converter and a punched paper tape to magnetic tape converter. Its output consisted of a12,800 character per second magnetic tape reader, a 120 card per minute card to tape converter, a 10 character per second character printer, a Uniprinter (a 600 line per minute high-speed line printer developed by Earl Masterson in 1954) and a 60 word per minute Rad Lab buffer. This was the first machine to use a buffered memory. It had 5,200 vacuum tubes, 18,000 crystal diodes, 300 relays and contained a mercury delay line memory that could hold 1,000 words 72 bits in length (11 decimal digits plus sign). The 8 ton, 25 by 50 feet machine consumed 125,000 Watts of power—31,250 times as much as a desktop computer (the average desktop consumes less than 400 Watts). It could perform 1,900 additions, 465 multiplications or 256 divisions per second. The machine also had a character set, similar to a typewriter keyboard, with capital letters. In 1956 a commercial UNIVAC computer was introduced that used transistors.

In 1943, the Massachusetts Institute of Technology (MIT) started the Whirlwind Project, under the supervision of Jay Forrester, for the U.S. Navy after determining that it was possible to produce a computer to run a flight simulator for training bomber crews. Initially, they attempted to use an analog machine but found that it was neither flexible nor accurate. Another problem was the typical batch-mode computers of the day were

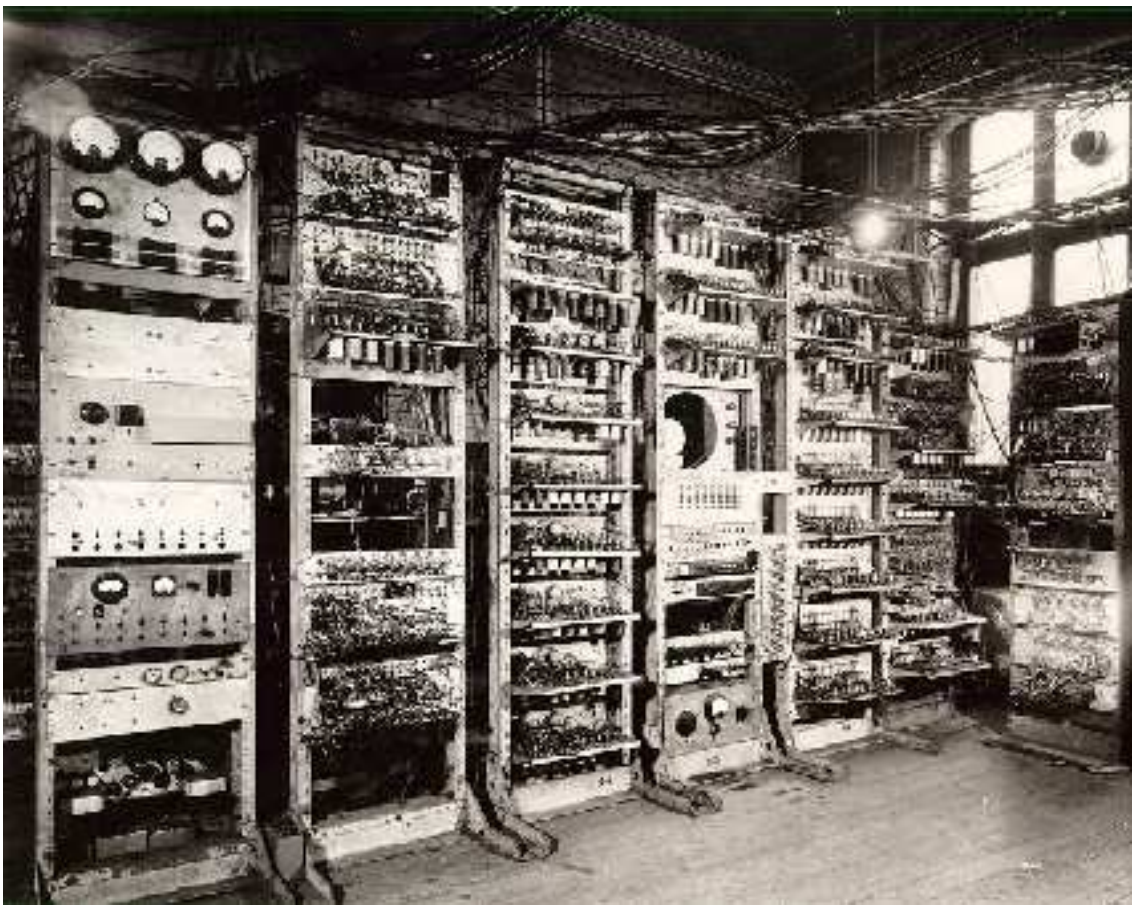


not computationally sufficient for time constrained processing because they could not continually operate on continually changing input. Whirlwind also required much more speed than typical computational systems. The design of this high-speed stored-program computer was completed by 1947 and 175 people started construction in 1948. The system was completed in three years, when the U.S. Air Force picked it up because the Navy had lost interest, renaming it Project Claude. This machine was too slow and improvements were implemented to increase performance. The initial machine used Williams tubes, cathode ray tubes that were used to store electronic data, which were unreliable and slow. Forrester expanded on the work of An Wang, who created the pulse transfer-controlling device in 1949. The product was magnetic core memory (upper left), which permanently stores binary data on tiny donut shaped magnets strung together by a wire grid. This approximately doubled the memory speed of the new machine, completed in 1953. Whirlwind was the world's first real-time computer and the first computer to use the cathode ray tube, which at this time was a large oscilloscope screen, as a video monitor for an output device.

The new machine was used in the Semi Automated Ground Environment (SAGE), which was manufactured by IBM and became operational in 1958. The picture on the right shows a SAGE terminal. This system coordinated a complex system of radar, telephone lines, radio links, aircraft and ships. It could identify and detect aircraft when they entered U.S. airspace. SAGE was contained in a 40,000 square foot area for each two-system installation, had 30,000 vacuum tubes, had a 4k by 32-bit word magnetic drum memory and used 3 megawatts of power. In 1958, the Whirlwind project was also extended to include an air traffic control system. The last Whirlwind-based SAGE computer was in service until 1983.[xxiii]

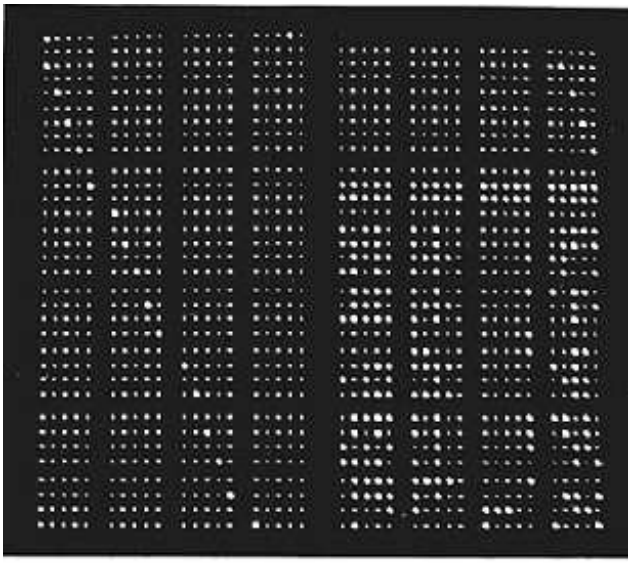


In 1946, work started on the Electronic Delay Storage Automatic Calculator (EDSAC), a serial electronic calculating machine, at Cambridge. It was contained in a 5 by 4 meter room, had 3000 valves, consumed 12,000 Watts and could perform 650 instructions per second at 500kHz. Its mercury ultrasonic delay line memory could 1024 words 17 bits in length (35-bit "long" digits could be contained by using two adjacent memory "tanks") and had an "Operating System" (called "initial orders") that was stored in 31 words in read-only memory". The input device consisted of a 6⅔ character per second 5-track teleprinter paper tape reader and output was performed on a 6⅔ character per second teleprinter. A commercial version of EDSAC, called LEO, which was manufactured by the Lyons Company, began service in 1953. Cambridge was the first university in the world to offer a Diploma in Computer Science, using EDASC, which was initially a one-year post graduate course called Numerical Analysis and Automatic Computing.[xxiv]



In 1948, at the University of Manchester in England, the Small Scale Experimental Machine, nicknamed the “Baby”, successfully executed its first program, becoming world's first stored-program electronic digital computer. Frederic C. Williams (1911 - 1977) and Tom Kilburn (1921 - 2001) built the machine to test the Williams-Kilburn Tube (type of memory composed of cathode vacuum tubes storing one bit of information on a cathode ray tube, illuminating a point on the screen that stays on) for speed and reliability, and to demonstrate the feasibility of a stored program computer. Its success prompted the development of the Manchester Mark I, a useable computer based on the same principals. The picture shows the “Baby” (replica), the shortest cabinet at the right, and the Mark I, the six taller cabinets.

The picture on the left shows Williams and Kilburn at the console of the Manchester Mark I. It was built in 1949 and could store data in addressable "line"s, holding one 40-bit number or two 20-bit instruction registers, and had two 20-bit address modifier registers, called "B-lines" (for modifying addresses in instructions), which functioned either as index registers or as base address registers. This Mark I was of historical significance because it is the first machine to include this index/base register in its architecture, which was a very important improvement. It was the first Random Access Memory computer. It could perform serial 40-bit arithmetic, with hardware add, subtract and multiply (with an 80-bit double-length accumulator) and logical instructions. The average instruction time was 1.8 milliseconds (about 550 additions per second), with multiplication taking much longer. It had a single-address format order code with about 30 function codes. The machine used two Williams tubes for its 128 words of memory. Each tube contained 64 rows with 40 points (bits) per row, which was two “page”s (A page was an array of 32 by 40 points). It also had a 128 page capacity drum-backing store, 2 pages per track, about 30 milliseconds revolution time on 2 drums (each drum could hold up to 32



tracks, i.e. 64 pages).

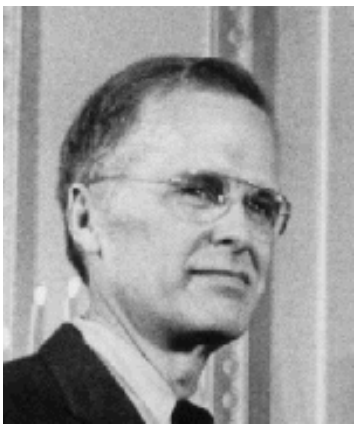
The machine's peripheral instructions included a "read" from a 5-hole paper tape reader, on which the code was normally entered, and "transfer" a page or track to or from a Williams-Kilburn Tube page or pair of pages in storage. It also had a bank of 40 (8 by 5) buttons that could be used to set the ones in a word in storage. There were also additional switches that controlled the operations of the Mark I. The current storage contents could be viewed on the machine's display tube, shown on the left, which was organized into 8 columns of 5-bit groups. There was a direct correspondence between the symbols, each made up of a 5-bit group, on the punched cards and the symbols on the display tube. The government awarded the contract to mass-produce Mark I computers to Ferranti Ltd., which was the world's first commercially available computer. Kilburn wrote the first electronically stored computer program for the Mark I and also established the world's first university computer science department at Manchester.[xxv]

There were substantial improvements in computer programming and user interface design as well as hardware architecture. John Mauchly (ENIAC and UNIVAC) developed Short Order Code, which is thought to be the first high-level language in 1949, for the Binary Automatic Computer (BINAC) computer. The BINAC, completed in 1949, was designed for Northrop Aviation and was the first computer to use a magnetic tape. In 1951, David Wheeler, Maurice Wilkes, and Stanley Gill introduced sub-programs and the "Wheeler jump", to implement them by moving to a different section of instructions and returning to the original section after the sub-program is finished. Maurice Wilkes also originated the concept of micro-programming, which is a technique for providing an orderly approach to designing the control section of a computer system.

In 1951, while working with the UNIVAC I mainframe, Betty Holberton (left) created the sort-merge generator, which was predecessor to the compiler and may have been the first useful program that had the capability of generating other programs for the UNIVAC I, and developed the C-10 instruction code, which controlled the its core functions. The C-10 instruction code allowed UNIVAC to be controlled by a control console (keyboard) commands instead of switches, dials and wires, which made the system much more useful and human friendly. The code was designed to use mnemonic characters to input instructions, such as 'a' for add. She later was the chairperson for the

committee that established the standards for the Common Business Oriented Language (COBOL).[xxvi]

In 1952, Grace Murray Hopper developed A-0, which is believed to be the first real compiler or an intermediary program that converts symbolic mathematical code into a sequence of instructions that can be executed by a computer. This allowed the use of specific call numbers assigned to the collected programming routines that were stored on magnetic tape, which the computer could find and execute. In the same year she developed a compiler for business use, B-0 (later renamed FLOW-MATIC) that could translate English terms and wrote a paper that described the use of symbolic English notation to program computers, which is much easier to use than machine code that was previously used. While working on the UNIVAC I, she encouraged programmers to reuse common pieces of code that were known to work well, reducing programming errors. She was on the CODASYL Short Range Committee to define the basic COBOL language design, which appeared in 1959 and were greatly influenced by FLOW-MATIC. COBOL was launched in 1960 and was the first standardized computer programming language for business applications. Various computer manufacturers and the Department of Defense supported development of the standard. It was intended to solve business problems, be machine independent and to be updated. COBOL has been updated and improved over the years, and is still used today. Hopper spent many years contributing to the standardization of compilers, which eventually led to international and national standards and validation facilities for many programming languages.[xxvii]



In 1956, John Backus and his IBM team created the first FORTRAN (short for FORmula TRANslation). The initial compiler consisted of 25,000 lines of machine code, which could be stored on magnetic tape. Backus and team wrote the paper "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN" to communicate their discovery and to show that scientists and mathematicians could program without actually understanding how the machines worked or without knowing assembly language. It works by using a software

device called a translator, which contains a parser to translate the high-level language that could be read by people to a binary language that can be executed on a computer. A later version of FORTRAN is still in use today, over 40 years later. Backus also developed a standard notation, Backus-Naur Form (BNF), to unambiguously and formally describe a computer language. BNF uses grammatical-type rules to describe a language.



In 1947, a major event occurred in electronics and computation. John Bardeen, Walter Brattain and William Shockley (pictured in order on left) announced that they developed the transistor for which they were awarded the Nobel Prize in 1956. This invention ushered in a new era in computers. First generation computers used vacuum tubes as their principal digital circuits. Vacuum tubes generated heat, consumed electrical power and quickly burned out, requiring frequent maintenance. They were also used in telecommunications to amplify long distance phone calls, which is the reason for this team's research. Transistors can switch and modulate electronic current, and are composed of a semi-conductor that can both conduct and insulate, such as germanium or silicon. The transistor can act as a transmitter by converting sound waves into electronic waves and a resistor by controlling electrical current. In 1954, Texas Instruments lowered the cost of production by introducing silicon transistors. The transistor brought about the second generation in computers by replacing vacuum tubes with solid-state components, which began the semiconductor revolution.[xxviii] Philco Corporation engineers developed the surface barrier transistor in 1954, which was the first transistor suitable for use in high-speed computers. In 1957, Philco completed the TRANSAC S-2000—the first large-scale, fully transistorized scientific computer to be offered as a manufactured product.[xxix]

In 1957, the Burroughs Atlas computer, constructed at the Great Valley Research Laboratory outside of Philadelphia, was one of the first to use transistors. The machine was developed for the America air defense system deployed during the 1950's and was the ground guidance computer for the Atlas intercontinental ballistic missile (ICBM). The first launch was in 1958. The system had two memory areas, one for data with 256 24-bit words and one for instructions with 2048 18-bit words. There were 18 Atlas computers constructed, costing $37 million.[xxx]

After the launch of Sputnik (NASA recreated model pictured on left) by the U.S.S.R. in 1957, The U.S. government responded by forming the Advanced Research Projects Agency (ARPA) to ensure technological superiority by expanding new frontiers of technology beyond immediate requirements. Initially ARPA's mission concerned issues, including space, ballistic missile defense, and nuclear test detection. The major contribution that ARPA made to computer technology was the Advanced Research Projects Agency Network (ARPANET).

In 1960, Paul Baran of the RAND Corporation published studies on secure communication technologies that would allow military communications to continue operations after a nuclear attack. He discovered two important ideas that outline the packet-switching principal for data communications:

1. Use a decentralized network having multiple paths between any two points, which allows single points of failure from which the system could automatically recover
2. Divide complete user messages into blocks before sending them into the network

In 1961, Leonard Kleinrock performed research on “store and forward” messaging, where messages are buffered completely on a switch or router, checksummed to find if an error exists in the message, and sent to the next location. In 1962, J.C.R. Licklider from MIT discussed the “Gallactic Network” concept in a series of memos. These computer network ideas represent the same type of general communication system as is used in the Internet. The same year that he wrote these memos, Licklider was working at ARPA and was able to convince others that this was an important idea. In 1966, Lawrence G. Roberts from MIT was brought in to head the APRANET project to build the network. Roberts’ "plan for the ARPANET" was introduced at a symposium in

1967, which included a time-sharing scheme using smaller computers to facilitate communication between larger machines as suggested by Wesley Clark. An updated plan was completed in 1968, which included packet switching. The contract to construct the network was awarded to Bolt, Beranek and Newman in early 1969. The first connected network consisted of four nodes between UCLA, the Stanford Research Institute, UCSB, and University of Utah. It was completed in December 1969. The ARPANET was the world's first operational packet switched network. Packet switching was a new concept that allowed more than one machine to access one channel to communicate with other machines. Previously these channels were switched and only allowed one machine to communicate with one other machine at a time. By 1973, the University College of London in England and the Royal Radar Establishment in Norway connect to the ARPANET, making it an international network.

With the advent of computer internetworking came new innovations to facilitate communication between machines. One innovation formulated by Robert Kahn and Vint Cerf was to make host computers responsible for reliability, instead of the network as was done in the initial ARPANET. This minimized the role of the network, which made it possible to connect networks and machines with different characteristics and, made the development of the Transmission Control Protocol (TCP)—to check, track and correct transmission errors and the Internet Protocol (IP)—to manage packet switching. The TCP/IP suite is arranged as a layered set of protocols, called the TCP/IP Stack, which defines each layers responsibilities in the connectionless transmission of data and interfaces that allow the passing of data between each layer. Because the interfaces between each layer are standardized and well defined, development of hardware and software is possible for different purposes, and from different architectures. The TCP/IP protocols replaced the Network Control Protocol (NCP), the original ARPANET protocol, and the military part of ARPANET was separated, forming MILNET, in 1983. The initial network restricted commercial activities because it was government funded.

In the early 1970's, message exchanges that were initially available on mainframe systems became available across wide area networks. In 1972, Ray Tomlinson introduced the "name@computer" addressing scheme to simplify e-mail messaging, which is still in use today. In 1972, the Telnet standard for terminal emulation over TCP/IP networks, which allows users to log onto a remote computer, was introduced. It enables users to enter commands on offsite computers, executing the as if they were using the remote systems own console. In 1973, the File Transfer Protocol (FTP) was developed to facilitate the long-distance transfer of files across computer networks. The Unix User Network (Usenet) was created in 1979 to facilitate the posting and sharing of messages, called "articles", to network distributed bulletin boards, called "newsgroups". In the mid 1980's the Domain Name System used Domain Name Servers to simplify machine identification. Instead of using a machines IP address, such as "10.192.20.128",

a user only need remember the machines domain name, such as “thismachine.net”. By 1982, commercial e-mail service was available in 25 cities and the term “Internet” was designated to mean a “connected set of computer networks”. In 1983, the complete change to TCP/IP created a truly global “Internet”.

National Science Foundation (NSF) became involved in ARPANET in the mid 1980’s. In 1986, the NSFNet Backbone was started to connect and provide access to supercomputers. In the late 1980’s, the Department of Defense stopped funding for ARPANET and the NSF assumed responsibility for long-haul connectivity in 1989. The first Internet Service Providers (ISP) companies also appeared, servicing regional research networks and providing access to email Usenet News for the public. The NSF initiated the connection of regional TCP/IP networks and the Internet began to emerge. In the 1990’s, commercial activity was allowed and the Internet grew rapidly. Eventually, this commercial activity created competition and commercial regional providers, called Network Access Points (NAP’s) took over backbones and interconnections, causing NSFNet to be dropped and the removal of all existing commercial restrictions.

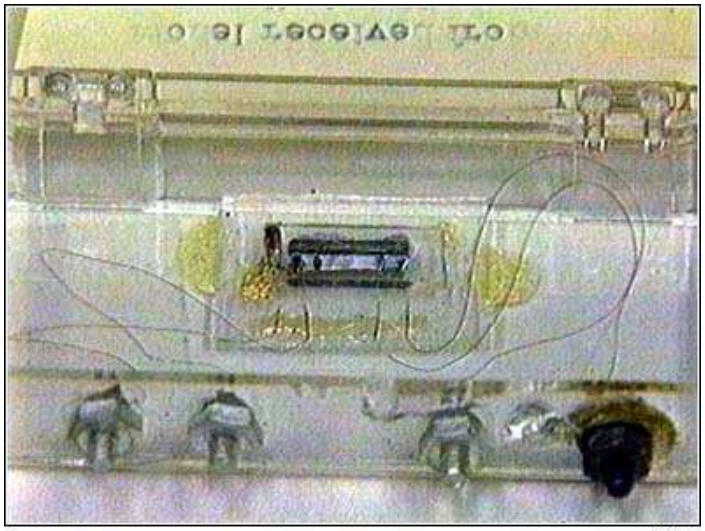
In 1989, Tim Berners-Lee invented the Uniform Resource Locator (URL) and Hypertext Markup Language (HTML), which were inspired by Vannevar Bush's "memex". The URL provides a simple way to find specific documents on the Internet by using the name of the machine, the name of the document file and the protocol to obtain and display the file. HTML is a method to set the format a document by embedding codes, which can also be used to designate hypertext—text that can be “clicked” on with a mouse pointer to cause some action or to retrieve another document. Eventually it was possible to place graphics and sound in documents, which started the World Wide Web (WWW), and many of the services that are now available on the Internet. By 1997, 150 countries and 15 million host computers were connected to the Internet, and 50 million people were using the World Wide Web. By 1990, approximately 9 million people will send over 2.3 billion e-mail messages.[xxxi]

In 1958, the ALGOrithmic Language (ALGOL) 58 high-level scientific programming language was formalized. It was designed to be a universal language by an international committee. It was the first attempt at software portability to provide a machine independent implementation. ALGOL is considered to be an important language because

it influenced the development of future languages. Almost all languages have been developed with "ALGOL-like" lexical and syntactic structures that have hierarchal, nested environment and control structures. ALGOL 60 had block structure for statements and the ability to call subprograms by name or by value. It also had if-then-else control statements for iteration and with recursive ability. ALGOL has a small number of basic constructs with a non-restricted associated type and rules to combine them into more complex constructs, of which some can produce values. ALGOL also had dynamic arrays wit variable specified subscript ranges, reserved words for key functions that could not be used as identifiers, and user defined data types to fit particular problems. A sample ALGOL source code "Hello World!" program from the Web site referenced for this information that runs on a Unisys A-series mainframe is:[xxxii]

```
BEGIN
FILE F (KIND=REMOTE);
EBCDIC ARRAY E [0:11];
REPLACE E BY "HELLO WORLD!";
WHILE TRUE DO
  BEGIN
  WRITE (F, *, E);
  END;
END.
```

As of 1959, more that 200 programming languages had been created.

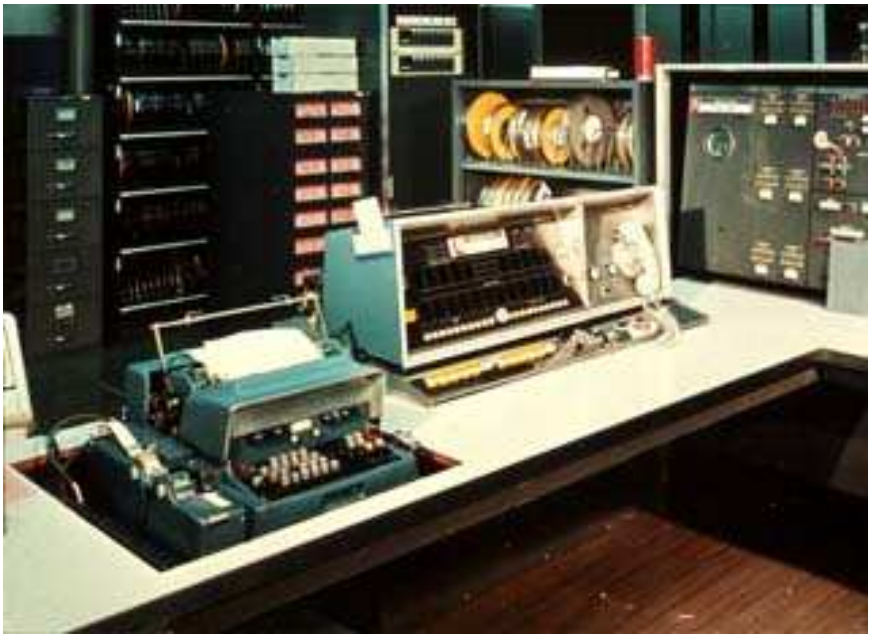


(CNN)

Between 1958 and 1959, both Texas Instruments and Fairchild Semiconductor Corporation were introducing integrated circuits (IC). TI's Jack Kirby, an engineer with a background in transistor-based hearing aids, introduced first IC (pictured left from CNN), which was based on a germanium semiconductor. Soon after, one of Fairchild's founders and research engineers, Robert Noyce, produced a similar device based on a silicon semiconductor. The monolithic integrated circuit combined transistors, capacitors, resistors and all connective wiring on a single semiconductor crystal or chip. Fairchild produced the first commercially available ICs in 1961. Integrated circuits quickly became the industry standard

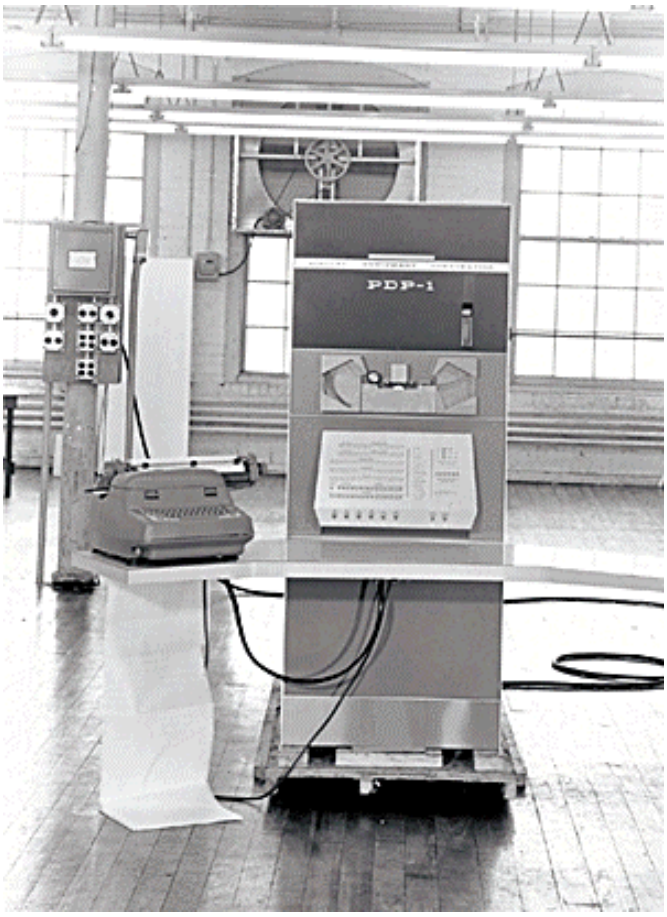
architecture for computers. Robert Noyce later founded Intel. Jack Kirby had commented:

"What we didn't realize then was that the integrated circuit would reduce the cost of electronic functions by a factor of a million to one, nothing had ever done that for anything before" [xxxiii]



In 1960, The Remington Rand UNIVAC delivered the Livermore Advanced Research Computer (LARC) computer to the University of California Radiation Laboratory, now called the Lawrence Livermore National Laboratory. This machine had four major cabinets that were approximately 20 feet long, 4 feet wide and 7 feet tall. One cabinet contained the I/O processor to route and control input and output, another had the computational unit to perform computational activity, and the last two contained 16K of ferrite core memory. There were also twelve floating head drums, rotating cylinders coated with a magnetic material, that were approximately 4 feet wide, 3 feet deep and 5 feet high, which were used as storage devices. Each drum could store 250,000 12-decimal-digit LARC words—almost 3 Megs on its 12 drums. There were also two independent controllers for read and write operations. There were also eight tape head units that could hold approximately 450,000 LARC words on each tape reel, deducting storage overhead. Its printer could print 600 lines per minute and had a 51 alphanumeric characters set. There was a punch card reader and a control console with toggle switches to control the system (pictured above). The LARC performed decimal mode arithmetic operations to 22 decimal digits and could perform 12x12 addition in 4 microseconds and 12x12 multiplication in 12 microseconds, with division taking a little bit longer. The machine used storage, shift and result registers to store information during repetitive calculations. LARC's hardware was difficult to maintain due to its

discrete nature, being comprised of a collection of transistors, resistors, capacitors and other electronic components.[xxxiv]



In November of 1960, Digital Equipment Corporation (DEC) started production of the world's first commercial interactive computer, the PDP-1 (left). The $120,000 machine's four cabinets measured approximately 8 feet in length. A DEC technical bulletin describes it as:

"...a compact, solid state general purpose computer with an internal instruction execution rate of 100,000 to 200,000 operations per second. PDP-1 is a single address, single construction, stored program machine with a word length of 18-bits operating in parallel on 1's complement binary numbers."

It had a 4000 18-bit word memory. It was the first computer with a typewriter keyboard and a cathode-ray tube display monitor. It also had a light pen, which made it interactive, and a paper punch output device. Producing 50 of these machines made DEC the world's first mass computer maker.[xxxv]

Between 1961 and 1962, Fernando Corbató of MIT developed Compatible Time-Sharing System (CTSS) as part of Project MAC, which was one of the first time-sharing operating systems that allowed multiple users to share a single machine. It was also the first system to have formatting text utility and one of the first to have e-mail capabilities. Louis Pouzin developed RUNCOM for CTSS, the precursor of UNIX shell script, which executed commands contained in a file and allowed parameter substitution. Multiplexed Information and Computing Service (Multics), the operating system that led to the development of UNIX, was also developed by project MAC. This system was the successor to CTSS and was used for multiple-access computing.[xxxvi]

In 1962, the Telstar I communications satellite was launched and relayed the first transatlantic television signals. The black and white image of an American flag was relayed from a large antenna in Andover, Maine to the Radome in Pleumeur-Bodou, France. This was the first satellite built for active communications. It demonstrated that a worldwide communication system was feasible. The satellite was launched by NASA from Cape Canaveral, Florida, weighed 171 pounds and was 34 inches in diameter. On the same day, the Telstar I beamed the first satellite long distance phone call. The satellite was in service until 1963. As of 2002, there were 260 active satellites in Earth's orbit.

In late 1962, the Atlas computer (left) entered service at the University of Manchester, England. This was the first machine to have pipelined instruction execution, virtual memory and paging, and separate fixed and floating-point arithmetic units. At the time it was the world's most powerful computer capable of about 200,000 FLOPS. It could perform the following arithmetic operations (approximate times):

- Fixed-point addition in 1.59 microseconds
- Floating-point add in 1.61 microseconds
- Floating-point multiply in 4.97 microseconds

The machine could timeshare between different peripheral and computing operations, was multiprogramming capable, had interleaved stores, had V-stores to store images of memory, had a one-level virtual store, had autonomous transfer units and ROM stores. It had an operating system called the "Supervisor" to manage the computers processing time and scheduling operations and could compile high-level languages. The machine had a 48-bit word size and a 24-bit address size. It could store 16K words in its main ferrite core memory, interleaving odd and even address. It had an additional 96K of storage in its four magnetic drum storage, which was integrated with the main memory using virtual memory or paging. It also accessed its peripheral devices through V-store addresses and extracode routines.[xxxvii]

In 1964, J. Kemeny and T. Kurtz, mathematics professors at Dartmouth College, developed the Beginner's All Purpose Symbolic Instruction Code (BASIC) as a simple to learn and interpret language that would serve to help students learn more complex and powerful languages, such as FORTRAN or ALGOL.[xxxviii] In the same year, IBM developed its Programming Language 1 (PL/1), formerly known as New Programming Language (NPL), which was the first attempt to develop a language that could be used for many application areas. Previously, programming languages were designed for a single purpose, such as mathematics or physics. PL/1 can be used for business and scientific purposes. PL/1 is a freeform language with no reserved keywords, has hardware independent data types, is block oriented, contains control structures to conditionally allow logical operations, supports arrays, structures and unions (and complex combinations of the three structures), and provides storage classes.[xxxix]

In 1962, Doug Englebart of the Stanford Research Institute published the paper: "Augmenting Human Intellect: A Conceptual Framework". His ideas proposed a device that would allow a computer user to interact with an information display screen by using a device to move a cursor on the screen—in other words, a mouse. The actual device, shown on the left, was invented in 1964.[xl] In the same year, the number of computers in the US grows to 18,000. In 1972, Xerox Palo Alto Research Center (PARC) Learning

Research Group developed Smalltalk. This forerunner of Mac OS and MS Windows was the first system with overlapping windows and opaque pop-up menus. In 1973, Alan Kay invented the "office computer", a forerunner of the PC and Mac. Its design was based on Smalltalk, with icons, graphics and a mouse. Kay stated at a 1971 meeting at PARC:

"Don't worry about what anybody else is going to do… The best way to predict the future is to invent it. Really smart people with reasonable funding can do just about anything that doesn't violate too many of Newton's Laws!" [xli]

In 1973, R. Metcalfe and researchers at Xerox PARC developed the experimental Alto PC that incorporates a mouse, graphical user interface and Ethernet. Within the same year, PARC's Charles Simonyi developed Bravo text editor, the first "What You See Is What You Get—type" (WYSIWYG) application. Metcalfe, later in the year, wrote a memo describing Ethernet as a modified "Alohanet", titled "Ether Acquisition". By 1975, Metcalfe developed the first Ethernet local area network (LAN). By 1979, Xerox, Intel and DEC had announced support for Ethernet. The Alto PC was officially introduced in 1981 with a mouse, built-in Ethernet and Smalltalk. The commercial version, available the same year, was named the Xerox Star and was the forst commercially available workstation with a WYSIWYG desktop-type Graphical User interface (GUI).

In 1964, Control Data Corp. introduced the CDC 6600 (left). It was designed by supercomputer guru Seymour Cray, had 400,000 transistors and was capable of 350,000 FLOPS. The 100 produced $7-10 million machines had over 100 miles of electrical wiring and a Freon refrigeration system to keep the system's electronics cool and were the world's first commercially successful supercomputer. The machine was also the first to have an interactive display the showed the graphical results of data, as it was processed in real-time.

Between 1964 and 1965, DEC introduced the PDP-8 (left)—the world's first minicomputer. It contained transistor-based circuitry modules and was mass-produced for the commercial market—the first computer sold as a retail product. During its initial offering at $18,000, it was the smallest and least expensive available parallel general-purpose computer. By 1973, the PDP-8, described as the "Model T" of the computer industry, was the best selling computer in the world. They had 12-bit words, usually with 4K words of memory, a robust instruction set and could run at room temperature.[xlii]

In 1965, Maurice V. Wilkes proposes the use of cache memory—a smaller, faster, more expensive type of memory that hold a copy of part of main memory. Access to entities in cache memory is much faster than that in main memory, which leads to better system performance. The same year, Intel founder Gordon Moore proposed that the number of transistors on microchips would double every year. The prediction was valid and came to be known as Moore's Law. Consider that a chip in 1964 that was 2½ cm^2 had ten components and a chip in 1970 of the same size had about 1000.

In 1967, Donald Knuth produced some of the work that would become "The Art of Computer Programming". He introduced the idea that a computer program's algorithms and data structures should be treated as different entities than the program itself, which has greatly improved computer programming. Volume 1 of The Art of Computer Programming was published in 1968.

In 1967, Niklaus Wirth began to develop the Pascal structured programming language. The Pascal Standard (ISO 7185) states that it was intended to:

- "make available a language suitable for teaching programming as a systematic discipline based on fundamental concepts clearly and naturally reflected by the language"
- "to define a language whose implementations could be both reliable and efficient on then-available computers"[xliii]

Pascal, based on ALGOL's block structure, was released in 1970. An example "Hello World!" Program in Pascal is:

```
Program Hello (Input, Output);
Begin
 Writeln ('Hello World!');
End.
```

In 1968, Burroughs introduced the first computers that used integrated circuits—the B2500 and the B3500. The same year Control Data built the CDC7600 and NCR introduced their Century series computer—both using only integrated circuits.

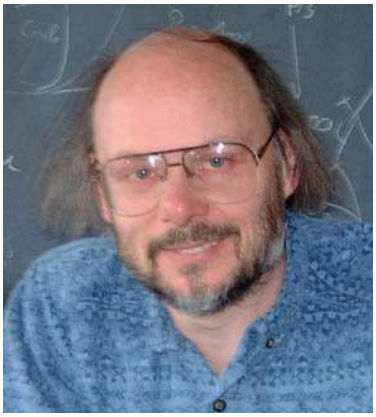
In 1968, the Federal Information Processing Standard created the "The Year 2000 Crisis" by encouraging the "YYMMDD" six-digit date format for information interchange. In 1968, the practice of structured programming started with Edsger Dijkstra's writings about the harm of the goto statement. This lead to wide use of control structures, such as the while loop, to control iterative routines in programs.[xliv] Between 1968 and 1969, NATO Science Committee held two conferences on Software Engineering, which is considered to be the start of this field. From the 1960's to the 1980's, there was a "software crisis" because many software projects had undesirable endings. Software Engineering arose from the need to produce better software, on schedule and within the anticipated budget. Essentially, Software Engineering is a set of diverse practices and technologies used in the creation and maintenance of software for diverse purposes.[xlv]

In 1969, Bell Labs withdrew support from Project MAC and the Multics system to begin development of UNIX. Kenneth Thompson and Dennis Ritchie began designing UNIX in the same year. The operating system was initially named Uniplexed Information and Computing System (UNICS) as a hack on Multics but was later changed. In the beginning, UNIX received no financial support from Bell Labs. Some support was granted to add text processing to UNIX for use on the DEC PDP-11/20. The text processor was named runoff, which Bell Labs used to record patent information, and later evolved into troff, the world's first publishing program with the capability of full typesetting. In 1973, it was decided to rewrite UNIX in C, a high level language, to make it easily modifiable and portable to other machines, which accelerated the development of UNIX. AT&T licensed use of this system to commercial, education and government organizations.

In 1973, Dennis Ritchie developed the C programming language. C is a high level programming language mainly to be used with UNIX. A sample "Hello World!" program in C is:

```
#include <stdio.h>

int main(){
printf ("Hello World!\n");
return 0;
}
```

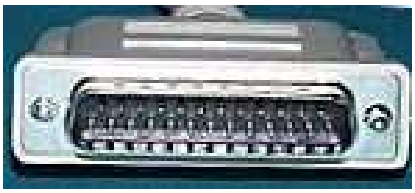


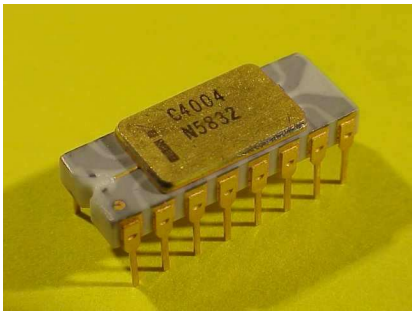
Later in 1983, Bjarne Stoustrup (right) added object orientation to C, creating C++, at AT&T Bell Labs. In 1995, Sun Microsystems released its object-oriented Java programming language, which was both platform independent and network compatible. Java is an extension of C++ and C++ is an extension of C.

By 1975, there were versions of UNIX using pipes for inter process communication (IPC). AT&T released a commercial version, UNIX System III, in 1982. Later, System V was developed by combining features from other versions, including U.C. Berkley's, Berkeley Software Distribution (BSD), which contributed the Vi editor and curses. Berkley continued to work on BSD the noncommercial version and added Transmission Control Protocol (TCP) and the Internet Protocol (IP), known as the TCP/IP suite, for network communication to the UNIX kernel. Eventually AT&T produced UNIX System V by adding system administration, file locking for file level security, job control, streams, the Remote File System and Transport Layer Interface (TLI) as a network application programming interface (API). Between 1987 and 1989, AT&T merged System V and XENIX, Microsoft's x86 UNIX implementation, into UNIX System V Release 4 (SVR4).

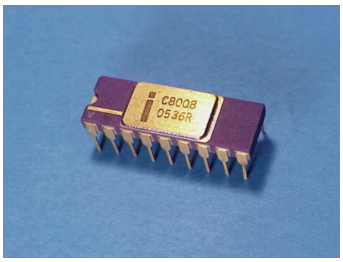
Novel bought the rights for UNIX from AT&T to in an attempt to challenge Microsoft's Windows NT, which caused their core markets to suffer. Novel sold the UNIX rights to X/OPEN, an industry consortium that defined a version of the UNIX standard, who later merged with OSF/1, another standard group, to form the Open Group. The Open Group presently defines the UNIX operating system.[xlvi]

In 1969, the RS-232 standard, commonly referred to as a serial port, for serial binary data interchange between Data terminal equipment (DTE) and Data communication equipment (DCE) was established.[xlvii]

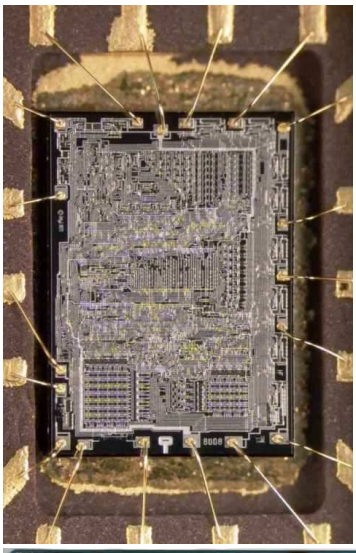


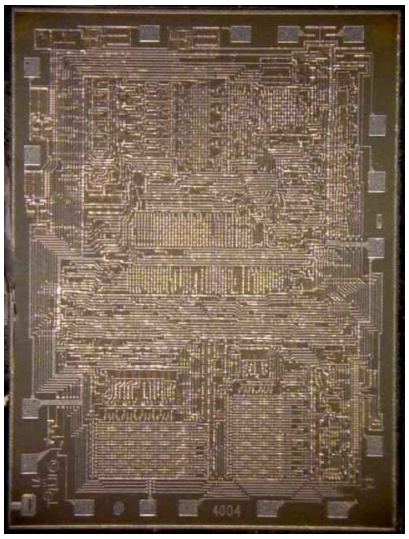


In 1970, RCA developed metal-oxide semiconductor (MOS) technology for fabricating integrated circuits, which made them smaller in size, cheaper and faster to produce. The first chips using large-scale integration (LSI) were produced in the same year, containing up to 15,000 transistors per chip. In 1971, Intel introduced the world's first mass produced, single chip, universal microprocessor, the Intel 4004 (left), which was

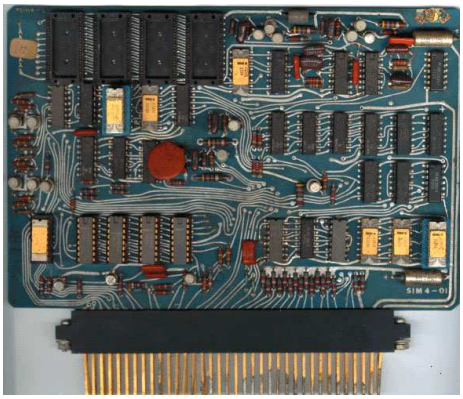


invented by Federico Faggin, Ted Hoff, Stan Mazor and their engineering team. It was a dual inline package (DIP) processor, which means that it had two rows of pins that were inserted into the motherboard. The microprocessor can be thought of as a "computer on a chip". All of the thinking parts of the computer, central processing unit (CPU), memory, input and output (I/O) controls, were miniaturized

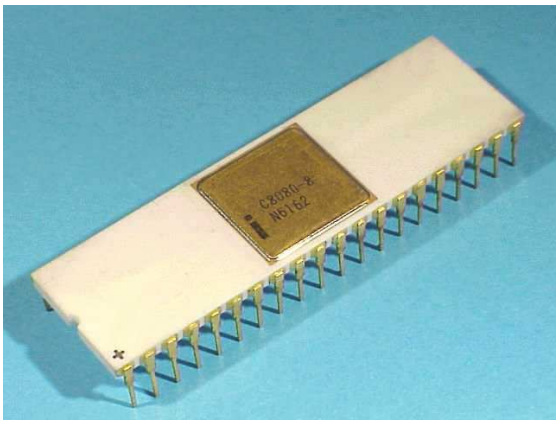




and condensed onto a single chip. The 4004 chip, based on the silicon-gated MOS technology, had more than 2,300 transistors in an area of 12 square millimeters, a 4-bit CPU that used 8-bit instructions, a command register, a decoder, decoding control, control monitoring of machine commands and an interim register. The chip ran at a speed of 108 kHz and could process 60,000 instructions



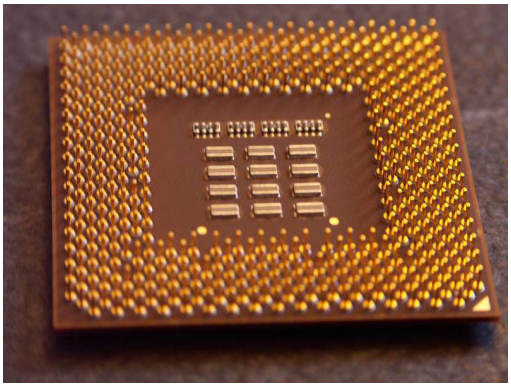
per second at a cost of $300. It had sixteen either 4-bit or 8-bit general-purpose registers and set of 45 instructions. It could address 1K of program memory and 4K of data memory. Later models had clock speeds of up to 740KHz. The picture on the lower left shows the 4004 motherboard and the picture on the right shows the chip die. The Pioneer 10 spacecraft, launched on March 2, 1972, used a 4004 processor and became the first spacecraft (and microprocessor) to enter the Asteroid Belt.[xlviii]



In 1972, Intel offered the 8008 chip (left), which was the world's first 8-bit microprocessor. The 8008 had 3300 transistors and even though its clock speed was 800 KHz it was slightly slower in instructions per second than the 4004 but because it was 8-bit, it could access more RAM and process data 3 to 4 times faster than the 4-bit chips. In 1974, Intel released the 8080 chip (left), which had a 16-bit address bus and an 8-bit data bus. It had a 16-bit stack pointer, a 16-bit program counter and seven 8-bit registers, of which some could be combined for 16-bit registers. It also had 256 I/O ports to ensure that devices did not interfere with its memory address space. It had a clock speed of 2 MHz, 64 KB of addressable memory, 48 instructions and vectored multilevel interrupts.

In 1978, Intel introduced the 8086 chip (left)—the first 16-bit microprocessor. This chip had 29,000 transistors, using a 3.0-micron die core design and 300 instructions. It had a 16-bit bus compatibility for communication with peripherals. The chips were available in 5, 6, 8, and 10 MHz clock speeds and had a 20-bit memory address space that could address up to 1 MB of RAM. Though the 8086 was available, IBM chose to use the 8088, the 8-bit version developed slightly later, because of the former chip's great expense.[xlix]

The Intel 80186, released in 1980, had a 16-bit external bus, an initial clock speed of 6 MHz and a 1.0-micron die. This chip was Intel's first pin grid array (PGA) offering, meaning that the pins on the processor were arranged into a matrix-like array with the pins around the outside edge (upper right). This popular chip was mostly used in imbedded systems and rarely used in PCs. This model required less external chips than its predecessors. It had an integrated system controller, a priority



interrupt controller, two direct memory access (DMA) channels (with controller), and timing circuitry (three timers). It replaced 22 separate VLSI and transistor-transistor logic (TTL) chips and was more cost efficient than the chips it replaced. In 1982, Intel developed the 80286 processor, which had 134,000 transistors, a 1.5-micron die, and could address up to 16 megabytes of memory. This microprocessor was the first to introduce the protected mode, which allowed the computer to multitask by running more than one program at a time by time-sharing the systems resources. Its initial models ran at 8, 10 and 12.5 MHz but later models ran as fast as 20 MHz. The 80386 processor was released in 1985 with 275,000 transistors, a 1.0-micron die, a 32-bit instruction and a 32-bit memory address space that could address up to four gigabytes of RAM. It had the ability to address up to 64 terabytes of virtual memory. The initial clock speeds were 16, 20, 25, and 33 MHz. It also had a feature called instruction pipelining, which allowed the processor to run the next instruction before finishing the previous instruction. It had a virtual real time mode that allowed more than one running session of real time programs, a feature that is used in multitasking operating systems. This chip also had a system management mode (SMM), which could power down various hardware devices to decrease power use. In 1989, Intel introduced the 80486 line of processors with 1.2 million transistors, a 1.0-micron die, and the same instruction and memory address size as the 386. This was the first microprocessor to have an integrated floating-point unit (FPU). Previously, CPUs had to have an external FPU, called a math coprocessor, to speed up floating-point operations. It also had 8 kilobytes of on-die cache, which stored predicted next instructions for pipelining. This saved an access to main memory, which is much slower than cache memory. Later 486 models could operate at greater speeds that the maximum system bus speed. The 486DX2/66 was a clock doubled 33 MHz to 66 MHz and the 486DX4/100 was clock a tripled 33 MHz to 100 MHz.

In 1993, Intel released the Pentium processor with 3.21 million transistors and a 0.8-micron die. Clock speeds were available from 60 to 200 MHz, with a 60 MHz processor capable of 100 MIPS. It had the same 32-bit address space as the 386 and 486 but had an external data bus width of 64 bits and a superscalar architecture (able to process two instructions per clock cycle), which allowed it to process instructions and data about twice as fast as the 486. Internally, this chip was actually two 32-bit processors chained together that shared the workload. It had two separate 8 KB caches (one data and one instruction cache) and a pipelined FPU, which could perform floating-point operations much faster than the 486. Later versions

of the chip had symmetric dual processing—the ability to have two processors in the same system.

In 1995, the Pentium Pro was released with 5.5 million transistors, a 0.6-micron die and a clock speed of up to 200 MHz. It was a reduced instruction set computer (RISC) processor. RISC processors have a smaller set of instructions than complex instruction set computer processors. The first computers were of CISC design to bridge semantic differences or gaps between low-level machine code and high-level programming languages, which reduced the size of computer programs and calls to main memory but did not necessarily improve system performance. The main idea with RISC is to build more complex instructions using a sequence of smaller, simpler instructions. Complex instructions have greater time and space overhead while decoding instructions, especially when microcode is used to decode macroinstructions. There is a high probability that the frequency of instructions to be processed will be smaller rather than larger. Limiting the number of instructions in a computer to a smaller optimized set can contribute to greater performance. The Pentium Pro could process three instructions per clock cycle and had decoupled decoding and execution, which allowed the processor to keep working on instructions in other pipelines if one of the pipelines stops to wait for an event. The standard Pentium would stop all pipelines until the event occurred. It also had up to 1 MB of onboard level-2 cache, which was faster than having the cache on the motherboard.

In 1997, Intel released the Pentium MMX series of processors with 4.5 million transistors, clock speeds up to 233 MHz and a 0.35-micron die size. The MMX had 57 additional complex instructions that aided the CPU in performing multimedia and gaming instructions 10 to 20 percent faster than processors without the MMX instruction set. The processor also had dual 16K level-1 cache and improved dynamic branch prediction, an additional instruction pipe and a pipelined FPU.

In 1993, Intel released the Pentium II, which had 27.4 million transistors and a 0.25-micron die. The Pentium II combined technology from both the Pentium Pro and the Pentium MMX. It had the Pro's dynamic branch prediction, the MMX instructions, dual 16K level-1 cache and 512K of level-2 cache. The level-2 cache ran at ½-speed and was not

attached directly to the processor, which yielded greater performance but not as much as if it were full-speed and attached. The most notable change was the single edge contact (SEC), called the "Slot 1", package design, which resembled a card more than it did a processor. Initial chips had a 66 MHz bus speed but later models had a 100 MHz bus. The bus speed is the maximum speed that the processor uses to access data in main memory.

In 1999, Intel released the Pentium III processor with 28 million transistors, a 0.18 die and a 450 MHz clock speed. This processor had 70 additional instructions that were extensions of the MMX set, called the SSE instruction set (also known as the MMX2 instruction set), which improved the performance of 3D graphics applications. Later versions of the Pentium III increased the bus speed to 133 MHz and moved the level-2 cache off of the board and onto the CPU core. Though Intel halved the memory to 256K, there was still a benefit to performance.

In late 2000, Intel introduced the Pentium IV with 42 million transistors, 0.13-micron die and a new NetBurst architecture to support future increases in speed. NetBurst consists of the Hyper Pipelined Technology, the Rapid Execution Engine, the Execution Trace Cache and a 400MHz system bus. The Hyper Pipelined Technology doubled the width of the data pipe from 10 to 20 stages, which decreased the amount of work per stage and allowed it to handle more instructions. A negative consequence of widening the data pipe is that it took longer to recover from errors. A newer and advanced branch predictor aided the chip in hedging against this propensity. The Rapid Execution Engine was the inclusion of two arithmetic logic units operating at double the speed of the processor, which was necessary to handle the doubled data pipe. The Execution Trace Cache was a new kind of cache that could hold decoded instructions until they are ready for execution. The chip has less level-1 cache, 8K, to decrease latency.[1]

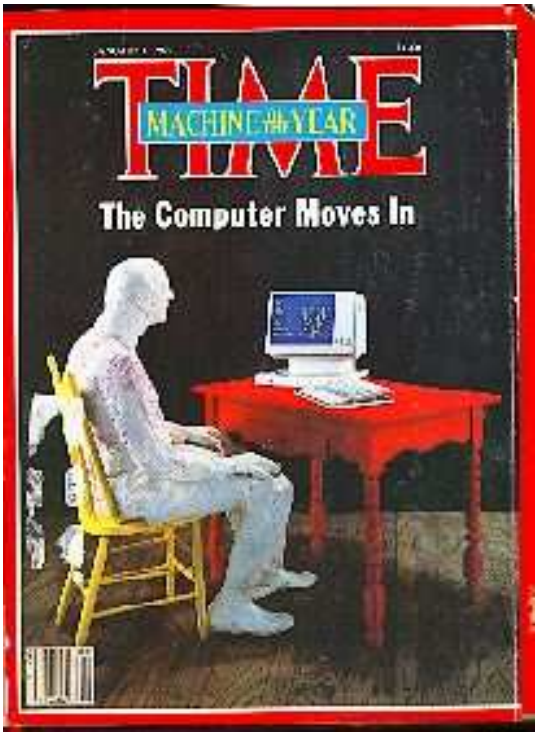
One of the ways Intel and other manufacturers have increased the speed and performance of CPUs was to decrease die size. This decreases the voltage needed to run the processor and increases clock speed. The functional part of a processor is actually a tiny chip with less than a third of a square inch of area within the external package shown in the preceding paragraphs. The chips are thinner than a dime and contain tens-of-millions of

electronic circuits and switches. The chips are constructed from semiconductor materials, such as gallium arsenide or most commonly silicon, which require certain conditions to conduct electricity. In the case of silicon, it is grown into a large crystal and sliced by precision saws into sheets, called wafers, which can hold many individual chips. Layers of various materials treated with a photosensitive material are built up on the surface of the wafer to form the foundation of the transistors and data pathways. A process called photolithography is used to process these wafers by copying the circuitry onto the layered materials on the wafer using a separate mask for each layer. Light is accurately focused through the masks, transferring the masks image onto the wafer, which causes a chemical reaction on the photosensitive material, fixing the circuitry. Another chemical is used to wash away the excess material. Sometime after the photolithography process is complete, the wafer is cut into small rectangular chips. The chips are installed into the CPU package by soldering the appropriate contacts on the chip with other circuitry and the pins that create the interface with the computer's motherboard.[li]

FIND MATERIAL ON ANALYTIC COMPLEXITY THEORY—1972

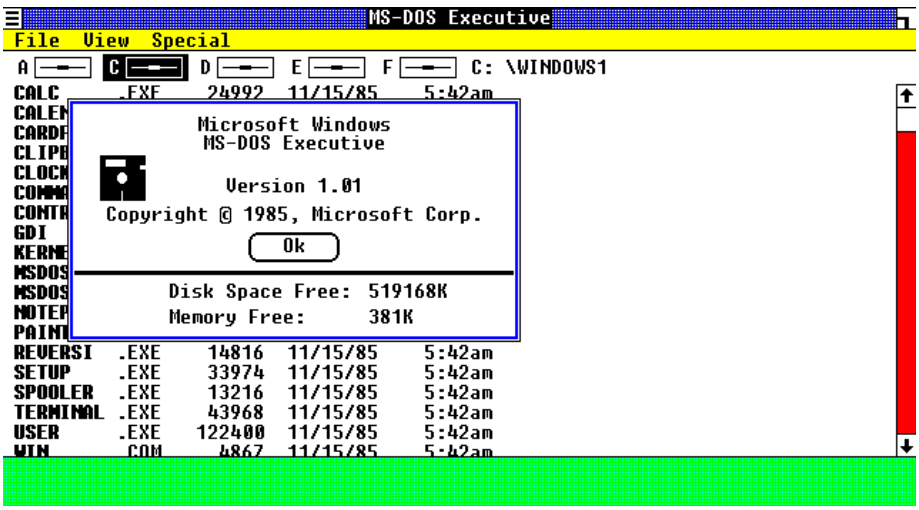
In 1975, Bill Gates and Paul Allen developed BASIC—the first microcomputer programming language. In 1977, Microsoft, Gates and Allen's newly founded company, released Altair BASIC for use on the Altair 8800. In 1980, Microsoft acquired the nonexclusive rights to an operating system, called 86-DOS, that was developed by a Seattle Computer Products' Tim Patterson. Microsoft had paid $100,000 to contract the rights from SCP to sell 86-DOS to an unnamed client. In 1980, IBM chose Microsoft product PC-DOS as the operating system for their new personal computer line.

The IBM PC became a mainstream corporate item when it was released in 1981. Microsoft bought all rights to 86-DOS in 1981, renaming it as MS-DOS. IBM's 5150 had a 4.77 MHz Intel 8088 CPU with 64K of RAM and 40K of ROM. It had a 5.25-inch, single-sided floppy drive, PC-DOS 1.0 installed and sold for $3000. IBM's new PC had an open architecture, which used off-the-shelf components. This was good for rapid and industry standard development but bad (for IBM) because other companies could obtain these components and build their own machines. In 1982, Columbia Data Products released the first IBM PC compatible "clone", called the MPC and Microsoft released an IBM compatible version operating system—MS-DOS v1.25, which could support 360K double-sided floppy

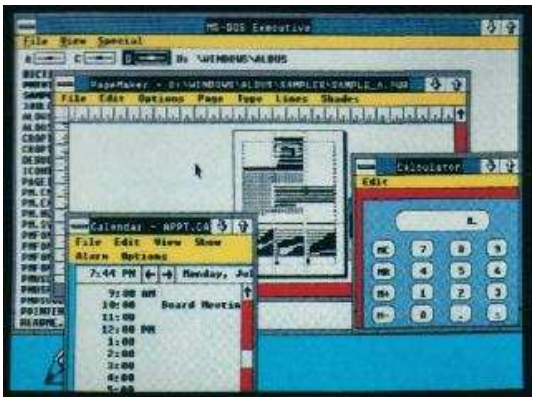


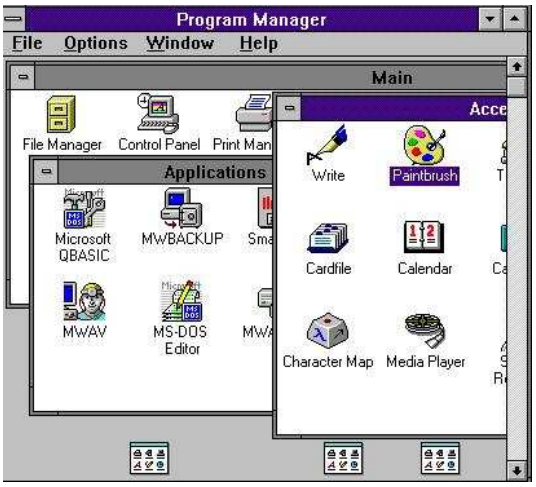
disks. The same year, Compaq introduces their first PC. The popularity of the PC caused sales to soar to 3,275,000 units in 1982, which was greater than ten times as many in 1981. The social impact of computers was so important that Time Magazine named the PC as its "Man of the Year" to be published on the cover of the January 1983 edition as the "Machine of the Year". By 1990, more that 54 million computers will be in use in the U.S. By 1996, approximately 66 percent of employees and 33 percent of homes have access to personal computers.

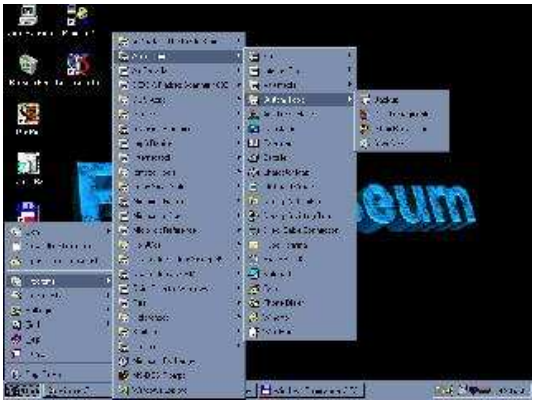
The initial MS-DOS offerings did not support hard disks. Version 2.0 in 1983 supported up to 10 MB hard disks and tree – structured file systems. Version 3.0 in 1984 supported 1.2 MB and hard disks larger than 10 MB and 3.1 had Microsoft network support. Version 4.0 in 1988 had graphical user interface support, a shell menu interface and support for hard disks larger than 32 MB. Version 5.0 in 1991 had a full-screen editor, undelete and unformat utilities, and task swapping. Version 6.0 in 1993 had DoubleSpace disk compression utility and sold over a million copies in 40 days. Version 7.0 of MS-DOS was included with Windows 95 in 1995.[lii]



In 1985, Microsoft introduced Windows 1.0(top left) with the promise of an easy-to-use graphical user interface, device independent graphics and multitasking support. A limited set of available applications lead to modest sales. Windows 2.0 (bottom left) was





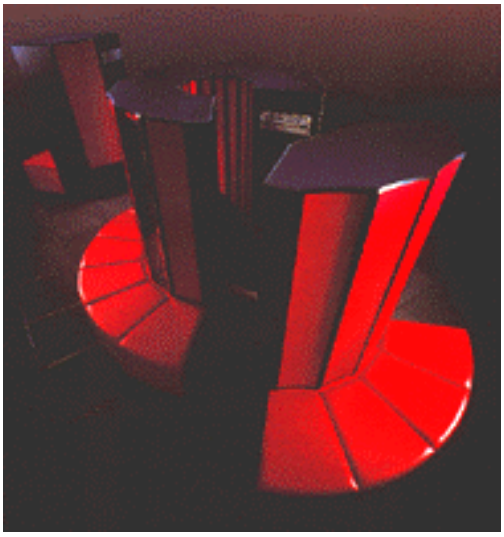


released in 1987 with two types available. One was for the 16-bit Intel 80286 microprocessor, called Windows/286. It added icons and overlapping windows with independently running applications. The other was for Intel's 32-bit line of 80386 microprocessors, which had all the functionality of the Windows/286 system but also had the ability to run multiple DOS applications, simultaneously. Windows 2.0 had much better sales due to the availability of software applications, including Excel, Word, Corel Draw!, Ami, Aldus PageMaker and Micrografx Designer. In 1990, Microsoft released Windows 3.0 (left) with a completely new interface and the ability to address memory beyond 640K without secondary memory manager utilities. Many independent software developers produced software applications for this environment, boosting sales to over 10,000,000 copies.

In 1994, Microsoft released Windows NT 3.1 with an entirely new operating system kernel. This system was intended for high-end uses, such as network servers, workstations and software development machines. Windows NT 4.0 was released later the same year and was an object-oriented operating system. In 1995, Microsoft introduced Windows 95 (left), which was a full 32-bit operating system. It had preemptive multitasking, multithreaded, integrated network, advanced file system. Though it included DOS 7.0, the Windows 95 OS assumed full control of the system after booting. In 1998, Windows 98 was released with enhanced Web support (the Internet Explorer browser was integrated with the OS), FAT32 for very large hard disk support, and multiple display support to use up to 8 video cards and monitors. It also had hardware support for DVD, Firewire, universal serial bus (USB) and accelerated graphics port (AGP). In 2000, Windows 2000 (formerly NT 5.0) was released and included many of the features of Windows 98,

including integrated Web support, and enhanced support for distributed file system. It also supported Internet, intranet and extranet platforms, active directory, virtual private networks, file and directory encryption, and installation of the W2K OS from a server located on the LAN.

1976, Cray Research developed the Cray-1 (left) supercomputer with vectorial architecture, which was installed at the Los Alamos National Laboratory. The $8.8 million machine could perform 160 FLOPS (world record at the time) and had an 8-megabyte (1 million words) main memory. The machines hardware contained no wires longer than four feet and had a "unique C-shape", which allowed integrated circuits to be very close together. In 1982, Steve Chen's and his research group built the Cray X-MP (right) by making architectural changes to the Cray-1, which contained two Cray-1 compatible pipelined processors and a shared memory (essentially two Cray-1 machines were linked together in parallel using a shared memory). This was the first use of shared-memory multiprocessing in vector supercomputing. The initial computational speedup of the two-processor X-MP over the Cray-1 was 300%—three times the computational speed by only doubling the number of processors. It was capable of 500 megaflops. This machine became world's most commercially successful parallel vector supercomputers. Chen commented that the X in X-MP stood for "extraordinary". The X-MP ran on UNICOS, which was Cray's first UNIX-like operating system. In 1985, the Cray-2 reached one billion FLOPS and had the world's largest memory at 2048 megabytes. In 1988, Cray produced the Y-MP, which was first supercomputer to "sustain" over one billion FLOPS on many of its applications. It had multiple 333 million FLOPS processors that could achieve 2.3 billion FLOPS.[liii]



In 1977, DEC introduced the 32-bit VAX11/780 computer (left), which was used primarily for scientific and technical applications. The first machine was installed at Carnegie Mellon University with other units installed at CERN in Switzerland and the Max Planck Institute in Germany. It could perform 1,000,000 instructions per second and was the first commercially available 32-bit machine.[liv]

In 1981, Motorola introduced one of the first 32-bit instruction microprocessor offerings from their 68000 line of processors. The chip has 32-bit registers and a flat 32-bit address space, which could access a specific memory location, instead of blocks of memory like the 8086. It had a 16-bit ALU but had a 32-bit address adder for address arithmetic. It had eight general-purpose registers and eight address registers. It used the last address register as a stack pointer and had a separate status register. It was initially designed as an embedded processor for household products but found its way into Amiga and Atari home computers and arcade computer games as a controller. It was also used in Apple Macintosh, Sun Microsystems and Silicon Graphics machines. The architecture of this chip was very similar to PDP-11 and VAX machines, which made it very compatible with programs written in the c language. The chip has been used by auto manufacturers as controllers as well as in medical hardware and computer printers because of its low cost. Updated models of the processor are still used today in personal digital assistants (PDAs) and Texas Instruments TI-89, TI-92 and Voyage 2000 calculators. In 1988, Motorola introduced the 88000 series processors, which were RISC-based, had a true Harvard architecture (separate instruction and data busses) and could perform 17 MIPS.[lv]

In 1985, Inmos introduced the transistor computer (transputer) with its concurrent parallel microprocessing architecture. Single transputer chips would have all the necessary circuitry to work by themselves or could be wired together to form more powerful devices from simple controllers to complex computers. Chips of varying power and complexity were available to serve a wide array of tasks. A low power chip might be

configured to be a hard disk controller and a few higher-powered chips might act as CPUs. These were the first general purpose chips to be specifically designed for parallel computing.

It was realized in the early 1980's that conventional CPUs would reach a performance limit. Even though advances in technology had miniaturized processor circuitry, packing millions of transistors on chips smaller than the size of a fingernail and had drastically increased computational speed, there was still a impenetrable barrier to conventional processor performance—the speed of light. Light in a vacuum travels at approximately 299,792,458 meters per second or approximately one foot in a nanosecond. This is the upper limit for the speed that electrons can travel within electrical equipment, which suggests that the clock speed limit for processors is about 10 GHz. We are almost half way to this limit and we realize that the speed of light is a limiting factor in the design of CPUs. The best way to ensure progress in computational performance is parallel processing.[lvi]

Parallel Processing

What is parallel processing?

Parallel processing is the concurrent execution of the same activity or task on multiple processors. The task is divided or specially prepared so that the work can be spread among many processors and yield the same result as if done on one processor but in less time. There is a variety of parallel processing systems. A parallel processing system can be a single machine with many processors or many machines connected by a network. The most powerful machines in the world are machines with hundreds or thousands of processors and hundreds of gigabytes of memory. These machines are called massively parallel processors (MPP). Many individual machines can cooperate to perform the same task in distributed networks. The combination of lower performance computers may exceed the power of a single high-performance computer, when the computational resources are comparable. The computational power of MPPs has been combined using the distributed system model to produce unprecedented performance.

Flynn's taxonomy classifies computing systems with respect to the two types of streams that flow into and out of a processor: instructions and data. These two types of streams can be conceptually split into two different streams, even if delivered on the same wire. The classifications, based on the number of streams of each type, are:

Single instruction stream/single data stream (SISD) systems have a single instruction processing unit and a single data processing unit. These are conventional single processor computers, also known as sequential computers scalar processors.

Single instruction stream/multiple data streams (SIMD) systems have a single instruction processing unit or controller and multiple data processing units. The instruction unit fetches and executes instructions until a data or arithmetic operation is reached. It then sends this instruction to all of the data processing units, which each perform the same task on different pieces of data, until all data is processed. These data processing units are either idle or all performing the same task as all other data processors. They cannot perform different tasks, simultaneously. Each of the data processors has a dedicated memory storage area. They are directed by the instruction processor to store and retrieve data to and from memory. The advantage of this system is the decrease in the amount of logic on the data processors. Approximately 20 to 50 percent of a single processor's logic is dedicated to control operations. The rest of the logic is shared by register, cache, arithmetic and data operations. The data processors have little or no control logic, which allows them to perform arithmetic and data operations much more rapidly. A vector or array processing machine is an example of an SIMD machine that distributes data across

all memories (possibly stores each cell of an array or each column of a matrix in a different memory area). These machines are designed to execute arithmetic and data operations on a large number of data elements very quickly. A vector machine can perform operations in constant time if the length of the vectors (arrays) does not exceed the number of data processors. Most supercomputers, used for scientific computing in the 1980's and 1990's, are based on this architecture.

Multiple instruction streams/single data stream (MISD) systems have multiple instruction processors and a single data processor. Few of these machines have been produced and have had no commercial success.

Multiple instruction streams/multiple data streams (MIMD) systems have multiple instruction processors and multiple data processors. There are a diverse variety of MIMD systems including those constructed from inexpensive off-the-shelf components to much more expensive interconnected vector processors, and many other configurations. Computers over a network that simultaneously cooperate to complete a single task are MIMD systems. Computers that have two or more independent processors are another example. A multiple independent processor machine has the ability to perform more than one task, simultaneously.[lvii]

There are three types of performance gains received from parallel processing solutions for the use of n processors:

- Sub-linear speedup is when the increase in speed is less than
    - i.e. five processors yields only 3x speedup
- Linear speedup is when the increase is equal to n
    - i.e. five processors yields 5x speedup
- Super-linear speedup is when the increase is greater than n
    - i.e. five processors yields 7x speedup

Generally linear or faster speedup is very hard to achieve because of the sequential nature of most algorithms. Parallel algorithms must be designed to take advantage of parallel hardware. Parallel systems may have one shared memory area, to which all processors may have access. In shared memory systems care must be taken to design parallel algorithms that ensure mutual exclusion, which protects data from being corrupted when operated on by more than one processor. The results from parallel operations should be determinate, meaning they should be the same as if done by a sequential algorithm. As an example, if two processors write to the same variable in memory such that:

- Processor 1 reads: x
- Processor 2 reads: x

- Processor 1 writes: x = x + 1
- Processor 2 writes: x = x – 1

Depending on the possible orderings of the reads and writes the resulting variable could be x–1, x+1 or x. This is a race condition and is an extremely undesirable because the result depends on chance. Synchronization primitives, such as semaphores and monitors, aid in the resolution of conflicts due to race conditions. The shared memory may be in a single machine if it has more than one processor or a distributed shared memory, where individual computers access the same memory area(s) located on another computer(s) on the network.

Parallel processors must use some means to communicate. This is done on the system buss and with shared memory in the case of a single computer with multiple processors. When multiple machines are involved, communication can be implemented over a network using either message passing or a distributed shared memory.

Cost is a very important consideration in distributed computing. A parallel system with n processors is cheaper to build than a processor that is n-times faster. For tasks that need to be completed quickly and can be performed by more than one thread of execution with minimal concurrency, parallel processing is an exceptional solution. Many high-performance or supercomputing machines have parallel processing architectures. The parallel implementations discussed in the remainder of this book will be based on distributed computing as opposed to single machines with multiple processors.

Existing Tools for Parallel Processing

The parallel programming systems discussed, PVM, MPI and Linda, are implemented with libraries of function calls that are coded directly into either C or Fortran source code and compiled. There are two primary types of communication used: message passing (PVM and MPI) and tuple space (Linda and Synergy). In message passing a participating process may send messages to any other process, directly, which is somewhat similar to inter-process communication (IPC) in the Linux/UNIX operating system. In fact, both message passing and tuple space systems are implemented with sockets in the Linux/UNIX environment. A tuple space is a type of distributed shared memory that is used by participating processes to hold messages. These messages can be posted or obtained by any of the participants. All of these programs function by the use of "master" and "worker" designations. The master is generally responsible to break the task into pieces and to assemble the results. The workers are responsible to complete their piece of the task. These systems are communicate over computer networks and typically have some type of middleware to facilitate cooperation between machines, such as the cluster discussed below.

Computer Clusters

Computer clusters, sometimes referred to as server farms, are groups of connected computers that form a parallel computer by working together to complete tasks. Clusters were originally developed in the 1980's by Digital Equipment Corporation (DEC) to facilitate parallel computing and file and peripheral device sharing. An example of a cluster would be a Linux network with some middleware software to implement the parallelism. Well established cluster systems have procedures to eliminate single point failures, providing some level of fault tolerance. The four major types of clusters are:

- Director based clusters—one machine directs or controls the behavior of the cluster and usually implemented to enhance performance
- Two-node clusters—two nodes perform the same part of the task or one serves as a backup in case the other fails to ensure fault tolerance
- Multi-node clusters—may have tens of clustered machines, which are usually on the same network
- Massively parallel clusters—may have hundreds or thousands of machines on many networks

Currently, the fastest supercomputing cluster is Earth Simulator at 35.86 TFlops, which is 15 TFlops faster than the second place machine. The main reason for cluster based

supercomputing, after performance, is cost efficiency. The third fastest supercomputing cluster is the 17.6 TFlop System X at Virginia Tech. It consists of 1100 dual processor Apple Power Macintosh G5s running Mac OS X. It cost a mere $5.2 million, which is 10 percent of the cost of much slower mainframe supercomputers.

The Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM), a software tool to implement a system of networked parallel computers, was originally developed by Oak Ridge National Laboratory (ORNL) in 1989 by Vaidy Sunderam and Al Geist. Version 1 was a prototype that was only used internally for research .PVM was later rewritten by University of Tennessee and released as Version 2 in 1991, which was used primarily for scientific applications. PVM Version 3, completed in 1993, supported fault tolerance and provided better portability. This system supports C, C++ and Fortran programming languages.

PVM allows a heterogeneous network of machines to function as a single distributed parallel processor. This system uses message-passing model as a means to implement the sharing of tasks between machines. Programmers use PVM's message passing to take advantage of the computational power of possibly many computers of various types in a distributed system, making them appear to be one virtual machine. PVM's API has a collection of functions to facilitate parallel programming by message passing. To spawn workers, the pvm_spawn() function is called:

```
int status = pvm_spawn(char* task, char** argv, int flag, char* where, int
ntask, int* tid);
```

where status is an integer that holds the number of tasks successfully spawned, task is the name of the executable to start, argv is the arguments for the task program, flag is an integer that specifies PVM options, where is the identifier of a host or system in which to start a process, ntask is an integer holding the number of task processes to start, and tid is an array to hold the task process ID's. To end another task process, use the pvm_kill() function:

```
int status = pvm_kill(int tid)
```

where status contains information about the operation, and tid is the task process number to kill. To end the calling task, use the pvm_exit() function:

```
int status = pvm_exit();
```

where status contains information about the operation. To obtain the task process ID of the calling function, use the pvm_mytid() function:

```
int myid = pvm_mytid();
```

where myid is an integer holding the calling function's task process ID. To obtain the task process ID of the calling function's parent, use the pvm_mytid() function:

```
int pid = pvm_parent();
```

where pid is an integer holding the parent function's task process ID. To send a message, the buffer must be initialized by calling the pvm_initsend() function:

```
int bufid = pvm_initsend(int encoding);
```

where bufid is the buffers ID number, and encoding is the method used to pack the message. To pack a string message into the buffer, use the pvm_pkstr() function:

```
int status = pvm_pkstr(char* msg);
```

where status contains information about the operation, and msg is a null terminated string. This function basically packs the array msg into the buffer. There are other functions to pack arrays of other data into the buffer. For a complete listing, see the PVM User's Guide listed in the references. To send a message use the pvm_send() function:

```
int status = pvm_send(int tid, int msgtag);
```

where status contains information about the operation, tid is the task process number of the recipient, and msgtag is the message identifier. To receive a message, use the pvm_recv() function:

```
int bufid = pvm_recv(int tid, int msgtag);
```

where bufid is the buffers ID number, tid is the task process number of the sender, and msgtag is the message identifier. This is a blocking receive. Entering "-1" as the tid value is a wildcard receive and will accept messages from all task processes. To unpack a buffer, use the pvm_upkstr() function:

```
int status = pvm_upkstr(char* msg);
```

where status contains information about the operation, and msg is a string in which to store the message. To compile and run a PVM application type:

```
[c615111@owin ~/pvm ]>aimk master slave
[c615111@owin ~/pvm ]>master
```

The amik command compiles the application and the executable name of the master executable runs the application. An example of a PVM "Hello worker—Hello master" application is below. It demonstrates the structure of a basic PVM program. The master program is:

```
// master.c: "Hello worker" program
#include <pvm3.h>
#define NUM_WKRS 3

main(){

  int status;        // Status of operation
  int tid[NUM_WKRS];// Array of task ID's all must be unique in system
  int  msgtag;       // Message tag to ID a message
  int flag = 0;      // Used to specify options for pvm_spawn
  char buf[100];     // Message string buffer
  char wkr_arg0 = 0;// Null argument to activate workers
  char** wkr_args;  // Array of args to activate workers
  char host[128];   // Host machine name

  // Set wkr_args to start worker program to address of wkr_arg0
  // which has been set to 0 (NULL)
  wkr_args = &wkr_arg0;

  // Get host machine name
  gethostname(host, sizeof(host));

  // Get my task ID and print ID and host name to screen
  printf("Master: ID is %x, name is %s\n", pvm_mytid(), host);

  // Spawn a program executable named "worker"
  // Will return the number of workers spawned on success or 0 on error
  // The empty string (fourth arg) requests any machine
  // Putting a name in this arg would request a specific machine
  status = pvm_spawn("worker", wkr_args, flag, "", NUM_WKRS, tid);

  // If spawn was successful it will return NUM_WKRS
  // since there are NUM_WKRS workers
  if(status == NUM_WKRS){

    // Label first message as 1
    msgtag = 1;

    // Put message in buffer
    sprintf(buf, "Hello worker from %s", host);

    // Initialize the send message operation
    pvm_initsend(PvmDataDefault);
```

```
    // Transfer the message to PVM storage
    pvm_pkstr(buf);

    // Send the message signal to all workers
    for(i=0; i< NUM_WKRS; i++)
      pvm_send(tid[i], msgtag);

    // Print messages sent to workers
    printf("Master: Messages sent to %d workers\n")

    // Get replies from workers
    for(i=0; i< NUM_WKRS; i++){

      // Execute a blocking receive to wait for reply from any (-1) worker
      pvm_recv(-1, msgtag);

      // Put the received message in the buffer
      pvm_upkstr(buf);

      // Print the message
      printf("Master: From %x: %s\n", tid, buf);

    }

    // Print end message
    printf("Master: Application is finished\n");

  }

  // Else the spawn was not successful
  else
    printf("Cannot start worker program\n");

  // Exit application
  pvm_exit();
}
```

The master program spawns a number of workers, sends the "Hello worker…" message and waits for a reply. After the reply is received, it is printed to screen and the master terminates. The worker program is:

```
// worker.c: "Hello Master" program
#include <pvm3.h>

main(){

  int ptid;       // Parents task ID
  int msgtag;     // Message tag to ID a message
  char buf[100];  // Message string buffer
  char host[128]; // Host machine name
  FILE* fd;       // File in which to write master's message

// Open file to store message
fd = fopen("msg.txt", "a");
```

```
  // Get host machine name
  gethostname(host, sizeof(host));

  // Get parents task ID
  ptid = pvm_parent();

  // Label first message as 1
  msgtag = 1;

  // Execute a blocking receive to wait for message from master
  pvm_recv(ptid, msgtag);

  // Put the received message in the buffer
  pvm_upkstr(buf);

  // Print the message to file
  fprintf(fd, "Worker: From %x: %s\n", ptid, buf);

  // Put reply message in buffer
  sprintf(buf, "Hello master from %s", host);

  // Initialize the send message operation
  pvm_initsend(PvmDataDefault);

  // Transfer the message to PVM storage
  pvm_pkstr(buf);

  // Send the message signal to master
  pvm_send(ptid, msgtag);

  // Close file
  fclose(fd);

  // Exit application
  pvm_exit();
}
```

The worker waits for the initial message from the master, writes the message to a file, sends a reply and terminates. The output on the master machine would resemble:

```
[c615111@owin ~/pvm ]>master
Master: ID is 0, name is owin
Master: Messages sent to 3 workers
Master: From 3: Hello master from saber
Master: From 1: Hello master from sarlac
Master: From 2: Hello master from owin
Master: Application is finished
```

All the workers output can be redirected to the master's terminal by running the application in PVM's console, which can be started by typing:

```
[c615111@owin ~/pvm ]>pvm
```

```
pvm>spawn -> master
```

Typing "pvm" at the command prompt activates the console and typing "spawn -> master" at the console prompt executes the application in console mode. The "->" causes all worker screen output to be printed on the masters terminal. At any point or time in a parallel application any executing PVM task (worker) may:

- Create or terminate other tasks
- Add or remove computers from the parallel virtual machine
- Have any of its process communicate with any other task's processes
- Have any of its process synchronize with any other task's processes

By proper use of PVM constructs and host language control-flow statements, any specific dependency and control structure may be employed under the PVM system. Because of its easy to use programming interface and its implementation of the virtual machine concept, PVM became popular in the high-performance scientific computing community. Currently it is not being developed but it made a significant contribution to modern distributed processing designs and implementations.[lviii]

Message Passing Interface (MPI/MPICH)

The Message Passing Interface (MPI) is a communications protocol that was introduced in 1994. It is the product of a community effort to define the semantics and syntax for a core set of message passing libraries for use by a wide variety of users and that could be used on a wide variety of MPP systems. MPI is not a standalone parallel system for distributed computing because it does not include facilities to manage processes, configure virtual machines or support input/output operations. It has become a standard for communication among machines running parallel programs on distributed memory systems. MPI is primarily a library of routines that can be invoked from programs written in the C, C++ or Fortran languages. Its differential advantages over older protocols are portability and performance. Its more portable because MPI has an implementation for almost every distributed system and faster because it is optimized for the specific hardware on which it is run. MPICH is the most commonly used implementation of MPI.

The MPI API has hundreds of function calls to perform various operations within a parallel program. Many of these function calls are similar to IPC calls in the UNIX operating system. Some of the basic MPI functions will be briefly explained and used in an example program. Before any MPI operations can be used in a program the MPI interface must be initialized with the MPI_Init() function:

```
MPI_Init(&argc, &argv);
```

where argc is the number of arguments and argv is a vector of strings, both of which should be taken as command line arguments because the same program will be used for both the master and worker processes in the example application. After initialization, a program must determine its rank by calling MPI_Comm_rank(), designated by process number, to determine if it is the master or a worker process. The master will be process number 0. The function call is:

```
MPI_Comm_rank(MPI_Comm comm, int* rank);
```

where comm is a communicator and is defined in MPI's libraries and rank is a reference pointer to an integer to hold this process' rank. It may also be necessary for an application to determine the number of currently running processes. The MPI_Comm_size() function returns this number. The function call is:

```
MPI_Comm_size(MPI_Comm comm, int* size);
```

where comm is a communicator and is defined in MPI's libraries and size is a reference pointer to an integer to hold the number processes. To send a message to another process the MPI_Send() function is used as such:

```
MPI_Send(void* msg, strlen(msg)+1, MPI_Datatype type, int dest, int tag,
MPI_Comm comm);
```

where msg is a message buffer, strlen(msg)+1 sets the length of the message and its null terminal, type is the data type of the message as defined by MPI's libraries, dest is an integer holding the process number of the destination, tag is an integer holding the message tag, and comm is a communicator and is defined in MPI's libraries. This is a blocking send and will wait for the destination to receive the message before executing further instructions. To receive a message the MPI_Recv() function is used as such:

```
MPI_Recv(void* msg, int size, MPI_Datatype type, int source, int tag, MPI_Comm
comm, MPI_Status* status)
```

where msg is a message buffer, is an integer holding the size actual size of the receiving buffer, type is the data type of the message as defined by MPI's libraries, source is an integer holding the process number of the source, tag is an integer holding the message tag, comm is a communicator and is defined in MPI's libraries, and status is the data about the receive operation. To end an MPI application session the MPI_Finalize() function is called:

```
MPI_Finalize();
```

which disables the MPI interface. To compile and run an MPI application type:

```
[c615111@owin ~/mpi ]>mpicc -o hello hello.c
[c615111@owin ~/mpi ]>mpirun –np 4 hello
```

The mpirun command activates a MPI application named "hello" with 4 processes (1 master and 3 workers) and the mpicc command is actually not a proprietary compiler. It is a definition that is equivalent a call to the cc compiler with the following arguments to access the proper libraries:

```
[c615111@owin ~/mpi ]>cc -o hello hello.c -I/usr/local/mpi/include\
-L/usr/local/mpi/lib -lmpi
```

An example of an MPI application is:

```
// hello.c program
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv){

  int my_rank;         // Rank of process
  int p;               // Number of processes
  int source;          // Rank of sender in loops
  int dest;            // Rank of receiver
  int tag = 50;        // Tag for messages
  char buf[100];       // Storage buffer for the message
  MPI_Status status;   // Return status for receive
  FILE* fd;            // File in which to write master's message

// Open file to store message
fd = fopen("msg.txt", "a");

  // Get host machine name
  gethostname(host, sizeof(host));

  // Initialize MPI application session
  // No MPI functions may be used until this is called
  // This function may only be called once
  MPI_Init(&argc, &argv);

  // Get my rank
  // Master's rank will be '0'
  // Worker's ranks will be greater than '0'
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  // Get the number of running processes
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  // If my_rank != 0, I am a worker
```

```
if (my_rank != 0){

  // Set source to '0' for master
  source = 0;

  // Receive message from master i
  MPI_Recv(buf, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);

  // Print the message to file
  fprintf(fd, "Worker: %s\n", buf);

  // Put reply in buffer
  sprintf(buf, "Hello master from %s number %d", buf, my_rank);

  // Set destination to '0' for master
  dest = 0;

  // Send the reply to master
  // Use strlen(buf)+1 to include '\0'
  MPI_Send(buf, strlen(buf)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);

}

// Else my_rank == 0 and I am the master
else{

// Get my task ID and print ID and host name to screen
printf("Master: ID rank %d, name is %s\n", my_rank, host);

  // Put reply in buffer
  sprintf(buf, "Hello worker from %s number %d", buf, my_rank);

  // Send messages to all workers
  for (dest=1; dest<p; dest++){

    // Send messages to workers
    MPI_Send(buf, strlen(buf)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);

    // Print message to screen
    printf("Master: Sent: %s to %d\n", buf dest);

  }

  // Get replies from all workers
  for (source=1; source<p; source++){

    // Receive reply from worker i
    MPI_Recv(buf, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);

    // Print message to screen
    printf("Master: Received: %s\n", buf);

  }

}
```

```
  // Close file
  fclose(fd);

    // Print end message
    printf("Master: Application is finished\n");

  // End MPI application session
  // No MPI functions may be called after this function is called
  MPI—Finalize();
}
```

The screen output on the master machine would resemble:

```
Master: ID rank 0, name is owin
Master: Sent: Hello worker from owin number 0 to 1
Master: Sent: Hello worker from owin number 0 to 2
Master: Sent: Hello worker from owin number 0 to 3
Master: Received: Hello master from saber number 3
Master: Received: Hello master from owin number 1
Master: Received: Hello master from sarlac number 2
Master: Application is finished
```

Linda

Linda is an environment and coordination language for parallel processing that was initially developed as a research project and a commercial product at Yale University by David Gelernter and Nicolas Carriero. Linda's design is based on a compromise between message passing and shared memory within a distributed parallel processing system. This system introduced the concept of a tuple space, which is a distributed shared memory area in which machines can communicate by reading, taking or putting tuples.

A single tuple space is created when the master program is executed. Tuples are similar to a vector data type but do not have specified primitive or structured data types contained within them. This allows any data to be stored in a binary format within the tuple space. Any combination of mixed data types can be placed not only into a tuple space but also in individual tuples within the space. Linda tuples may have a maximum of 16 fields, which are separated by commas. Entries in the tuple space are identified by names or numerical values in the tuple's data rather than as an address in local machines. An example of a tuple space entry with 3 fields is:

```
("string", 123, 45.678);
```

which contains a character string, an integer and a floating point number, respectively. There are two kinds of tuples in Linda: active tuples, also called live or process tuples, are tuples that are under active evaluation, and passive tuples, also called data tuples, are

entries in the tuple space similar to the example above. Active tuples are created with the eval() function. The function call:

```
eval("worker", worker());
```

would create a tuple entry with "worker" in the first field and spawn a new process that will immediately call the worker() function. Passive tuples are created and added to the tuple space with the Linda's out() function. The function call:

```
out("string", 123, 45.678);
```

would create the tuple and add it to the tuple space.

Data can be either read or removed from the tuple space. A template is used to retrieve a tuple from the tuple space by matching a pattern in the fields of a tuple's fields. The following conditions must be met to match a template to a tuple:

1. The template and tuple both must have the same number of fields.
2. The template and tuple both must have the same types, values, and length of all literal values in corresponding fields.
3. The template and tuple both must have matching types and lengths of all formals in the corresponding fields.

A read operation, using the rd() function, leaves the tuple for other processes to access. The function call:

```
rd("string", 123, ? A);
```

reads a three entry tuple that has "string" as its first element and 123 as its second. The data in the third element is placed in the A variable. The in() function gets and removes an entry from the tuple space. The function call:

```
in("string", 123, ? A);
```

gets a three entry tuple that has "string" as its first element and 123 as its second. The data in the third element is placed in the A variable and the entry is removed from the tuple space.

Programming for a tuple space is similar to programming for shared memory because all participating processes share it. However it is also similar to message passing because entries are posted and taken from it. The major benefit of this system is that participants can enter and leave the system without formerly announcing an arrival or departure.

They can also take messages, data or tasks from the tuple space at their own pace, which can balance the workload, giving more work to machines capable of greater performance, and decrease the overall duration of a given task. Tuple spaces and load balancing will be discussed further in later sections.

It should also be noted that Linda tuple spaces do not observe a first in first out (FIFO) structure. Reading or retrieving an entry may not necessarily obtain the oldest entry, which may cause programming errors if this structure is assumed. Linda parallel programs are written with both the master and worker programs in the same source file. The master function is the main function and the worker is a named function. Linda has its own built in compiler to compile the executable. To compile and execute a distributed network application type:

```
[c615111@owin ~/linda ]>clc -o hello hello.cl
[c615111@owin ~/linda ]>ntsnet hello
```

The clc command activates Linda's compiler and the ntsnet command executes the hello program as a network application. An example of a Linda master or main function for the "Hello worker—Hello Master" application is:

```
// hello.cl program
#define NUM_WKRS 3

real_main(int argc, char* argv){

  int i;          // Loop counter
  int hello();    // Function declaration
  char buf[100];  // Message string buffer
  char host[128]; // Host machine name

  // Get host machine name
  gethostname(host, sizeof(host));

  // Print master's name
  printf("Master: Name is %s\n", host);

    // Put message in buffer
    sprintf(buf, "Hello workers from %s", host);

  // Put the message in the tuple space
  out("message", buf);

  // Start the workers
  for (i=0; i< NUM_WKRS; i++)

    // Start an active tuple (a worker process)
    eval("worker", worker(i));

  // Get all workers' reply from tuple space
```

```
  for (i=0; i< NUM_WKRS; i++){

    // Get reply and remove from tuple space
    in("reply", ? buf);

    // Print reply to screen
    printf("Master: %s\n", buf);

  }

  // Print end message to screen
  printf("Master: Application is finished\n");

  // End the master
  return(0);
}
```

An example of a worker function is:

```
// The worker function
worker(int i){

  char buf[100];  // Message string buffer
  char host[128]; // Host machine name

  // Get host machine name
  gethostname(host, sizeof(host));

  // Read the message from tuple space
  rd("message", ? buf);

  // Print the message to screen
  printf("Worker: %s number %d got %s\n", host, i, buf);

    // Put message in buffer
    sprintf(buf, "Hello master from %s number %d", host, i);

  // Put reply in tuple space
  out("reply", buf);

  // Print end message to screen
  printf("Worker: %s finished\n");

  // End the worker
  return(0);

}
```

Linda prints both the master and workers' output to the master's screen. The screen output on the master machine would resemble:

```
[c615111@owin ~/fpc01 ]>ntsnet hello
Master: Name is owin
```

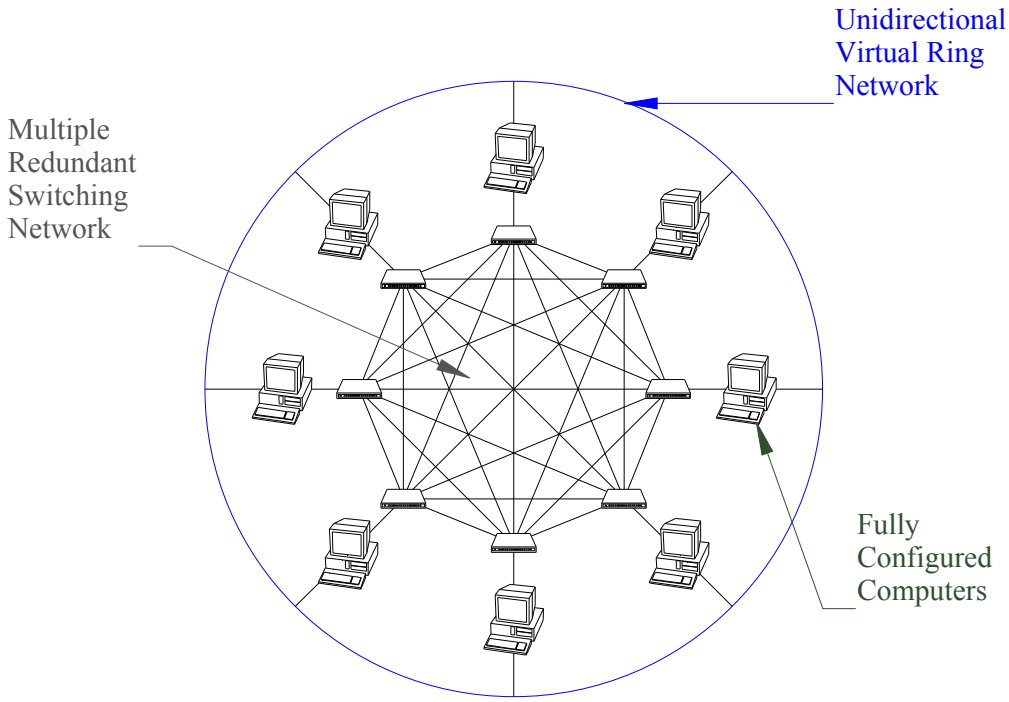
```
Worker: saber number 1 got Hello workers from owin
Worker: owin number 0 got Hello workers from owin
Worker: owin finished
Worker: sarlac number 2 got Hello workers from owin
Master: Hello master from sarlac number 2
Worker: saber finished
Worker: sarlac finished
Master: Hello master from saber number 1
Master: Hello master from owin number 0
Master: Application is finished
```

It should also be noted that global variables in Linda applications are not transferred to workers. Using global variables will have unpredictable results.[lix]

Parallel Programming Concepts

Stateless Parallel Processing (SPP)

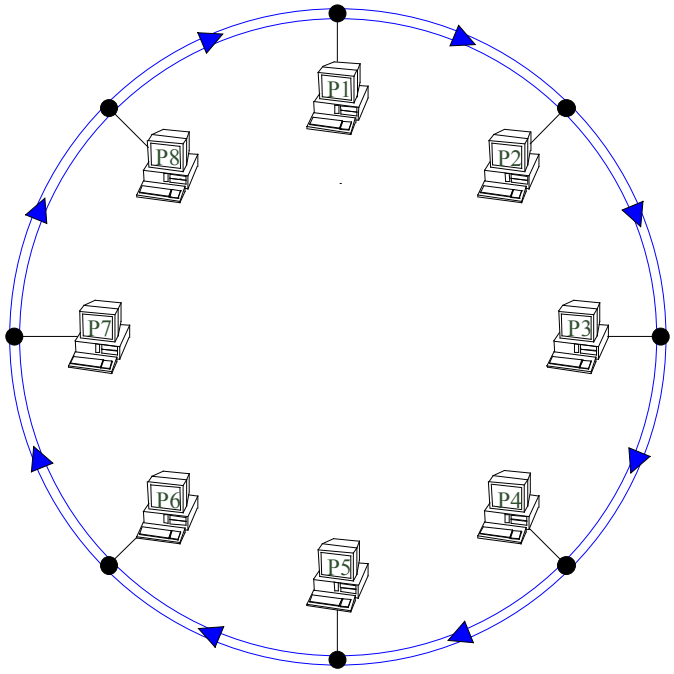
The Stateless Parallel Processing architecture is comprised of "fully configured computers" connected by a "multiple redundant switching network" that form a "unidirectional virtual ring network", as shown below. Multiple direct paths are provided from each node to every other node. Redundancy allows for scalable performance and fault tolerance.



The Stateless Parallel Processing Architecture

Please note that the unidirectional "virtual" network is implemented through the multiple redundant switching network's hardware and is not an actual physical ring. Each computer might have only one network interface adapter card. Each node on the virtual ring is aware of every other node because each maintains a current list of all participating nodes. Each node can also detect and isolate faulty nodes. The SPP virtual ring's responsibility is limited to tuple queries and SPP backbone management. Tuple data is transmitted directly from point to point. This ring also provides full bandwidth support for multicast communication through the network, where all nodes can access multicast

messages. The diagram below shows a conceptual representation of a unidirectional virtual ring, where the arrows may represent possibly a single multicast message that all nodes can acquire. The multiple switch network can transport a massive amount of data between machines.

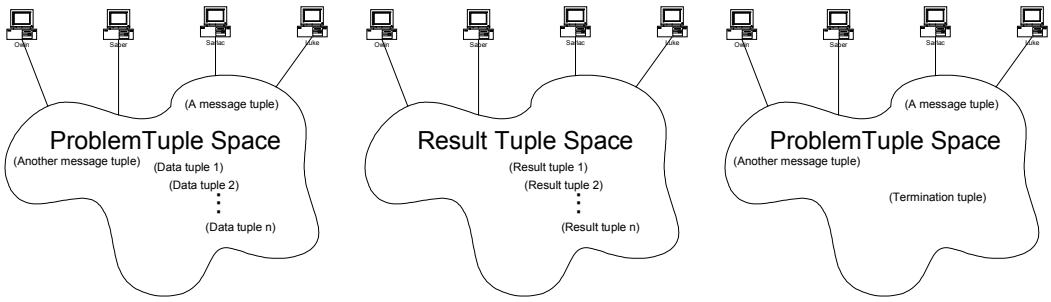


The Unidirectional Virtual Ring Configuration

The tuple space model allows participating processes to acquire massages from a current tuple space without temporal restrictions. Processes can take messages when they are ready without causing a work stoppage, unlike communication methods that uses a blocking send. In this design, tuples flow freely through the network from process to process. Each process will perform a part of the task by taking work date tuples from the tuple space at its own pace. The processes are purely data driven and will activate or continue processing only when it receives required data. There are no explicit global state controls in this "stateless" system, which ensures fault tolerance. If a process fails the system can recover because the data can be renewed in the tuple space and taken by another worker process.

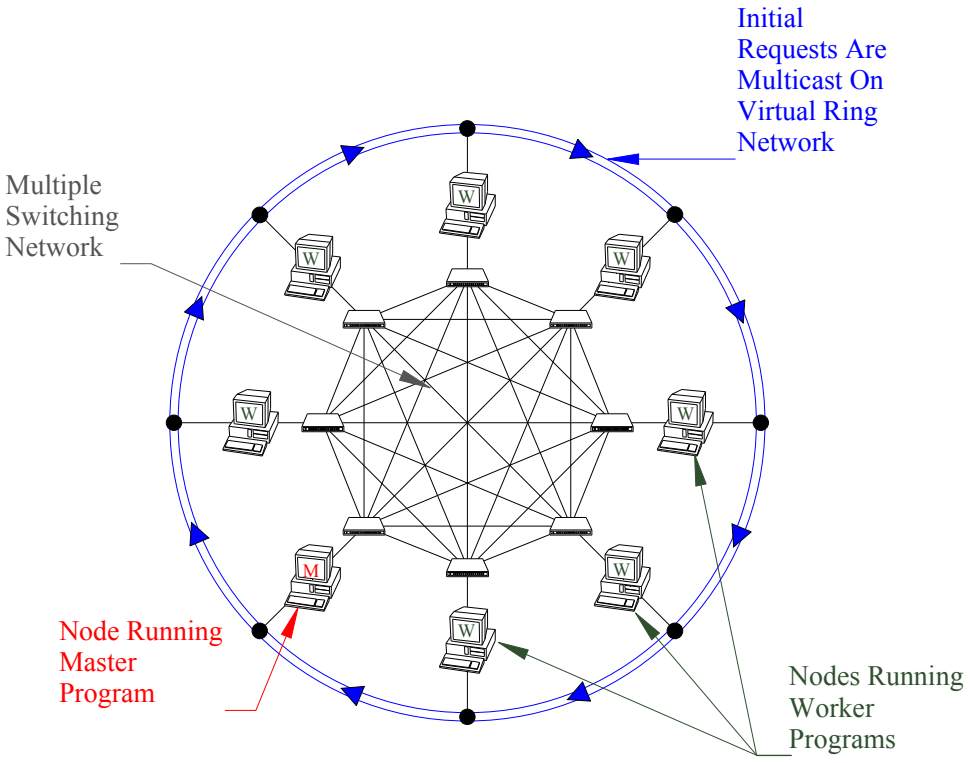
SPP applications use a parallel processing model called "scatter and gather", involving master and worker processes. A master process is the application controller for the worker processes. In a single task, single pass application, it divides the task into n subtasks, places the work data tuples in a tuple space, collects the completed subtasks

from a tuple space, and directs the workers to terminate when all of the results are received. The three diagrams below show possible contents during an applications execution.



The left-most diagram shows a problem tuple space, where work data is stored, after messages to workers and work data tuples have received. The center shows a result tuple space, where the master will receive completed subtasks. The right-most diagram shows a problem tuple space with a termination tuple, also called a poison pill, which instructs the workers to terminate. Notice that the message tuples remain in the tuple space and that the data tuples are removed. This is because the messages were accessed by a read operation and the data tuples were accessed by a take operation. If the terminal message is accessed by a take operation, it must be replaced so that the next worker can access it. This scenario assumes a parallel system that can create multiple tuple spaces, such a synergy. If the system is limited to one, then it depends more heavily on name pattern matching of tuples.

The master program with its accompanying tuple spaces can reside on any participating node. The worker processes take work tuples from the tuple space that match a tuple query, put the results into the result tuple space, until all work is completed, and terminate when they get the terminate message tuple from the master. The diagram below shows a possible master-worker configuration. It should be noted that the master machine generally has both a master process and a worker process. Otherwise a valuable system resource would be wasted because the master machine would be idle between receiving results.



The SPP Architectural Support

Stateless Machine (SLM)

A stateless machine (SLM) is a fully implemented stateless parallel processing system. An SLM should provide an API that offers a robust but easy to use interface with the system's functionality. It should have a fault tolerance facility to recover from dropped hosts and lost data. The network structure should offer high efficiency and high performance. The locations of processes should be transparent for all participating processes in the application, meaning that the system should handle communication between machines and not be directly noticeable to running programs. The workload should be balanced between the participating processes, where each process is kept busy until all work is complete.

Linda Tuple Spaces Revisited

As previously mentioned, the tuple space was first defined in the Linda distributed parallel programming implementation as a method of multi-machine inter-process coordination. It's easiest to think of a Linda tuple space as a buffer, a virtual bag or a public repository that cooperating processes from different computers can put tuples in, or read and get tuples from. It's a type of distributed shared memory, where any process can access any tuple, regardless of its storage location. A tuple space is not a physical shared memory. It is a logical shared memory because processes have to access it through an intermediary or tuple handling process. The API only makes the tuple space appear to be physically shared memory. The computers, though physically dispersed, must be part of some distributed system. The machines can communicate with each other without really being aware that any of the other machines exist, other than the data passed through the tuple space. Heterogeneous data types can be stored in tuples and differently structured tuples can be placed in the tuple space. Hence, all of the following data types:

```
char name[4] = {"Bob"};
int number = 12;
double fraction = 34.56;
```

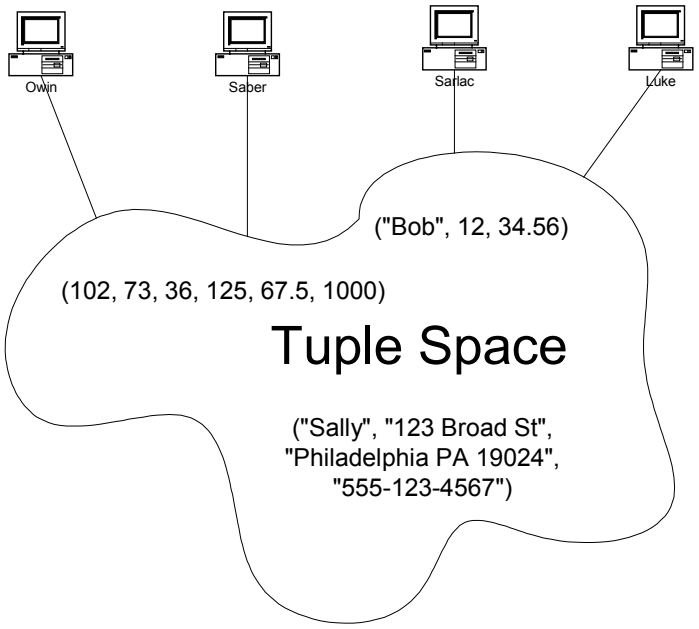
can be placed in the same tuple:

```
(name, number, fraction)
```

and all of the following tuples:

```
(name, number, fraction)
(102, 73, 36, 125, 67.5, 1000)
("Sally", "123 Broad St", "Philadelphia PA 19024", "555-123-4567")
```

can be placed in the same tuple space.



Tuples are placed in and retrieved from tuple spaces by function calls, previously described, that match a pattern from a template. A template is essentially a tuple that is used to express a pattern. The template:

(? A, 12, ? B)

where A is a string and B is a double, matches:

(name, number, fraction) = (“Bob”, 12, 34.56)

However, this template will not match the other tuples in the example above. The general rules for a Linda tuple were stated previously. This is called an associative memory because elements or tuples in the memory are accessed by associating them, synonymously, with a pattern in their content as opposed to being referenced by a memory address or physical location.

Active tuples in Linda are based on the generative communication model, where dynamically spawned processes are turned into data upon completion of their task. The eval(“worker”, worker()) function will leave a tuple in the tuple space with two fields from the called worker function:

```
worker(){
  // perform task
```

```
  return 0;
}
```

will place a tuple with the name assigned from the process that spawned the worker function in the first field (in this case "worker") and the return value of the worker function. All tuples placed by the worker into the tuple space will be accessible by all other processes even after the worker terminates. The tuple from the example above after the eval() function returns would be:

("worker", 0)

Since the concept was pioneered at Yale, many languages have been implemented using variants of Linda's tuple space model, including LiPS, ActorSpaces, TSpace, PageSpaces, OpenSpaces, Jini/Javaspaces, Synergy, etc.

Theory and Challenges of Parallel Programs and Performance Evaluation

Basic Logic

Logic is the study of the reasoning of arguments and is both a branch of mathematics and a branch of philosophy. In the mathematical sense, it is the study of mathematical properties and relations, such as soundness and completeness of arguments. In the philosophical sense, logic is the study of the correctness of arguments. A logic is comprised of an informal language coupled with model-theoretic semantics and/or a deductive system. The language allows the arguments to be stated, which is similar to the way we state our thoughts in written or spoken languages. The semantics provide a definition of possible truth-conditions for arguments and the deductive system provides inferences that are correct for the given language.

This section introduces formal logics that can be used as methods to design program logic and prove that the logic is sound. Systems based on propositional logic have been produced to facilitate the design and proofs for sequential programs. However, these systems were inadequate for concurrent applications. Variations of temporal logic, which is based on modal logic, are used to evaluate the logic of concurrent programs.

Propositional Logic

Symbolic logic is divided into several parts of which propositional calculus is the most fundamental. A proposition, or statement, is any declarative sentence, which is either true or false. We refer to true (T) or false (F) as the truth-value of the statement.

"1 + 1 = 2" is a true statement.
"1 + 1 = 11" is a false statement.
"Tomorrow will be a sunny day" is a proposition whose truth is yet to be determined.
"The number 1" is not a proposition because it is not a sentence.

Simple statements are those that represent a single idea or subject and contain no other statements within. Simple statements will be represented by the symbols: p, q, r and s. If p stands for the proposition: "ice is cold", we denote it as:

p: "ice is cold",

which is read as:

p is the statement “ice is cold”.

The following is an example of a simple statement assertion and negation.

p	assertion	p is true if p is true or p is false if p is false.
¬p	negation	¬p is false if p is true or ¬p is true if p is false.

Then for the true statement: p: “ice is cold”, ¬p is the statement that “ice is not cold”, which is false.

A compound statement is made up of two or more simple statements. The simple statements are known as components of the compound statement. These components may be made up of smaller components. Operators, or connectives, separate components. The sentential connectives are disjunction (∨, pronounce as OR), conjunction (∧, pronounce as AND), implication (→, pronounce as IF) and equivalence (↔, pronounce as IF AND ONLY IF). These are called sentential because they join statements, or sentences, into compound sentences. They are binary operators because they operate on two components or statements. Equivalence statements (p↔q) are also called biconditionals, and implication statements (p→q) are also called conditionals. In the p → q conditional statement, the "if- clause" or first statement, p, is called the antecedent and the "then-clause" or second statement, q, is called the consequent. The antecedent and consequent could be compounds in more complicated conditionals rather than the simple statements shown above. These terms are used for all the binary operators listed above. Negation (¬) is called a unary operator because it only operates on one component or statement. The following define the conditions under which components joined with connectives are true; otherwise they are false:

p∨q	disjunction	either p is true, or q is true, or both are true
p∧q	conjunction	both p and q are true
p→q	implication	if p is true, then q is true
p↔q	equivalence	p and q are either both true or both false

The statements:

p: “ice is cold”
q: 1 + 1 = 2
r: “water is dry”
s: 1 + 1 = 11

under conjunction:

p∧q is true because “ice is cold” is true and “1 + 1 = 2” is true
p∧r is false because “ice is cold” is true and “1 + 1 = 11” is false
s∧q is false because “1 + 1 = 11” is false and “1 + 1 = 2” is true
r∧s is false because “water is dry” is false and “1 + 1 = 11” is false

All meaningful statements will have a truth-value. The truth-value of a statement designates the statement as true T or false F. The statement p is either absolutely true or absolutely false. If a compound statement’s truth-value can be determined in its entirety based solely on its components, the compound statement is said to be truth-functional. If a connective constructs compounds that are all truth-functional, the connective is said to be truth-functional. Using these conditions it is possible to build truth-functional compounds from other truth-functional compounds and connectives. As an example: if the truth-values of p and of q are known, then we could deduce the truth-value of the compound using the disjunction connective, p∨q. This establishes that the compound, p∨q, is a truth-functional compound and disjunction is a truth-functional connective. A truth table contains all possible truth-values for a given statement. The truth table for p is:

p
T
F

because the simple statement p is either absolutely true or absolutely false. The following is the truth table of p and q for the five previously mentioned operators:

p	q	¬p	¬q	p∨q	p∧q	p→q	p↔q
T	T	F	F	T	T	T	T
T	F	F	T	T	F	F	F
F	T	T	F	T	F	T	F
F	F	T	T	F	F	T	T

Parentheses ( ) are used to group components into whole statements. The whole compound statement p∧q can be negated by grouping it with parentheses and negating the group ¬(p∧q). The table below shows all negated truth-values for the operators previous table.

p	q	¬(¬p)	¬(¬q)	¬(p∨q)	¬(p∧q)	¬(p→q)	¬(p↔q)
T	T	T	T	F	F	F	F
T	F	T	F	F	T	T	T
F	T	F	T	F	T	F	T
F	F	F	F	T	T	F	F

To avoid an excessive number of parentheses in statements, there is a standard for operator precedence. This simply means the order in which operations are performed. Negation has precedence over conjunction and conjunction has precedence over disjunction. The statement:

$\neg p \vee q$ is $(\neg p) \vee q$ not $\neg(p \vee q)$

and

$\neg p \vee q \wedge r$ is $((\neg p) \wedge q) \vee r$

A truth table will have 2^n rows, where n is the number of distinct simple statements in the whole statement. The first truth table for p had only two rows and the previous two had four rows. If p, q and r were under consideration, there would be eight rows. To find which values for p, q, and r will evaluate to true for $P(p, q, r) = \neg(p \vee q) \wedge (r \vee p)$, construct a truth table for the statement. Start by placing true values in the top row and false values in the next from the bottom row for one instance of each unique simple statement as shown below. The last row is to maintain the steps performed by operator precedence and parentheses. Mark all simple statements step 1.

¬	(p	∨	q)	∧	(r	∨	p)
	T		T		T		
	F		F		F		
	1		1		1		1

Then assume all F's are 0's and all T's are 1's, and count up the table from 0 to 7 in binary. Then copy values to all other duplicate simple statements.

¬	(p	∨	q)	∧	(r	∨	p)
	T		T		T		T
	T		T		F		T
	T		F		T		T
	T		F		F		T
	F		T		T		F
	F		T		F		F
	F		F		T		F
	F		F		F		F

	1		1		1		1

This holds all combinations of F's and T's relative to the three simple statements. Remember the pattern in the columns and you wont have to count next time. Next mark the second set columns to be evaluated by precedence and fill in the truth-values. Because of the parentheses, the next columns will be the third and seventh.

¬	(p	∨	q)	∧	(r	∨	p)
	T	T	T		T	T	T
	T	T	T		F	T	T
	T	T	F		T	T	T
	T	T	F		F	T	T
	F	T	T		T	T	F
	F	T	T		F	F	F
	F	F	F		T	T	F
	F	F	F		F	F	F
	1	2	1		1	2	1

Negation has precedence over conjunction. Hence the first column is the negation of the third. To find the truth-values for conjunction, consider the highest values in the last row on each side, which is column one on the left and column seven on the right.

¬	(p	∨	q)	∧	(r	∨	p)
F	T	T	T	F	T	T	T
F	T	T	T	F	F	T	T
F	T	T	F	F	T	T	T
F	T	T	F	F	F	T	T
F	F	T	T	F	T	T	F
F	F	T	T	F	F	F	F
T	F	F	F	T	T	T	F
T	F	F	F	F	F	F	F
3	1	2	1	4	1	2	1

The statement is only true for P(p, q, r) = P(F, F, T).

Again if p, q and r were under consideration, values for p, q, and r will evaluate to true for Q(p, q, r) = (p→q)∧[(r↔p)∨(¬p)], construct a truth table for the statement. Also note that brackets [ ] and braces { } can be used to differentiate compound groupings up to three levels.

(p	→	q)	∧	[(r	↔	p)	∨	(¬	p)]
T	T	T	T	T	T	T	T	F	T
T	T	T	F	F	F	T	F	F	T
T	F	F	F	T	T	T	T	F	T

T	F	F	F	F	F	T	F	F	T
F	T	T	T	T	F	F	T	T	F
F	T	T	T	F	T	F	T	T	F
F	T	F	T	T	F	F	T	T	F
F	T	F	T	F	T	F	T	T	F
1	2	1	4	1	2	1	3	2	1

There are three types of propositional statements that can be deduced from all truth-functional statements:

- If the truth-value column for the table has a mixture of T's and F's, the table's statement is called a contingency.
- If the truth-value column contains all T's, the statement is called a tautology.
- Lastly, if the truth-value column contains all F's, the statement is called a contradiction.

The following logical equivalences apply to any combination of statements used to create larger compound statements. The p's, q's and r' s can be atomic statements or compound statements.

The Double Negative Law	$\neg(\neg p) \equiv p$
The Commutative Law for conjunction	$p \wedge q \equiv q \wedge p$
The Commutative Law for disjunction	$p \vee q \equiv q \vee p$
The Associative Law for conjunction	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
The Associative Law for disjunction	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
DeMorgan's Law for conjunction	$\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$
DeMorgan's Law for disjunction	$\neg(p \wedge q) \equiv (\neg p) \vee (\neg q)$
The Distributive Law for conjunction	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
The Distributive Law for disjunction	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
Absorption Law for conjunction	$p \wedge p \equiv p$
Absorption Law for disjunction	$p \vee p \equiv p$
Conditional using negation and disjunction	$p \rightarrow q \equiv (\sim p) \vee q$
Equivalence using conditionals and conjunction	$p \leftrightarrow \equiv (p \rightarrow q) \wedge (q \rightarrow p)$

Predicate Calculus

Another part of symbolic logic is predicate calculus, which is built from propositional calculus. Predicate calculus allows logical arguments based on some or all variables under consideration. Consider the following arguments, which cannot be expressed in propositional logic:

All dogs are mammals

Fido is a dog
Therefore, Fido is a mammal

The three statements:

p: All dogs are mammals
q: Fido is a dog
r: Fido is a mammal

are of the form:

$$\begin{array}{l} p \\ \underline{q} \\ \therefore r \end{array}$$

can be independently evaluated under propositional logic but cannot be evaluated to derive the conclusion "r: Fido is a mammal" because "therefore" ('$\therefore$') is not a legitimate propositional logic operator. We need to expand propositional calculus and set theory to make use of the predicate calculus.

We use the universal quantifier $\forall$, which means for all or for every, to establish a symbolic statement that includes all of the things in a set X that we are considering as such:

$$\forall x[Px \rightarrow Qx]$$

The brackets define the scope of the quantifier. This example is read "For every variable x in set X, if Px then Qx". Applied to the example above, we could reword the statement "All dogs are mammals" by letting Px be: "if x is a mammal" and Qx be "then x is a mammal". We have:

"For all x, if x is a dog, then x is a mammal".

This is called a statement form and will become a statement when x is given a value. Let f = Fido. A syllogism is a predicate calculus argument with two premises sharing a common term.

$$\begin{array}{l} \forall x[Px \rightarrow Qx] \\ \underline{Pf} \\ \therefore Qf \end{array}$$

The predicate P means “is a dog” and Q means “is a mammal”. The conclusion states that because Fido is a dog, Fido is a mammal. If we negate the quantifier as such:

$\neg\forall x[Px\rightarrow Qx]$

The statement becomes:

“Not every dog is a mammal”.

Which sounds ridiculous but the statement is permissible by predicate logic. We can change this to:

$\forall x[Px\rightarrow\neg Qx]$

Which translates to:

“Some dogs are not mammals”.

Mathematical statements can be constructed using propositional calculus. The statement:

“If a integer is less than 10, then it is less than 11”

This statement can be converted using the universal quantifier so that is true for every integer x ($x \in N$) less than 10 as such:

$\forall x \in N\ [(x<10) \rightarrow (x<11)]$.

Which translates to:

“For every x that is an integer, if x is less than 10, then x is less then 11”.

If a logical statement is to be constructed for one or more members of a set but not necessarily all, we can use the existential quantifier, $\exists$, which means “there exists” of “for some”. The statement:

“Some lawyers speak the truth”,

would be restated as:

“There exists a lawyer that speaks the truth”.

If we let Px be "x is a lawyer" and Qx be "x speaks the truth", we have:

$\exists x\,[Px \wedge Qx]$,

which states that at least one lawyer speaks the truth. Quantifiers can be applied to more then one variable in a statement.

Let P be "is a shoe in my closet", where x is a right shoe and y is a left shoe. Then:

$\forall x, \exists y[Px \wedge Py]$,

is a symbolic representation of the statement: "For every right shoe in my closet, there exists a left shoe". A mathematical statement would be:

$\exists z \in N\,[x = y \times z], x \in N, y \in N$,

which states that there exists an integer z, such that integer x is divisible by integer y.[lx]

Modal Logic

Modal logic extends the capabilities of traditional logic to include modal expressions, which contain premises such as "it is necessary that…" or "it is possible that…". Modal logic is the study of deductive behavior of expressions based on necessary and/or possible premises. Modal logic can also be defined as a family of related logical systems that include logics for belief and temporal related expressions. The table below contains some common symbols and definitions used in the modal logic family:

Logic	Symbols	Expressions Symbolized
Modal Logic	□	It is necessary that …
	◊	It is possible that …
Deontic Logic	O	It is obligatory that …
	P	It is permitted that …
	F	It is forbidden that …
Temporal Logic		
	G	It will always be the case that …
	F	It will be the case that …
	H	It has always been the case that …
	P	It was the case that…
Doxastic Logic		
	Bx	x believes that …

A popular weak modal logic K, conceived by Saul Kripke, .defines three operators: "negation" ($\neg$), "if…then…" ($\rightarrow$), and "it is necessary that…" ($\Box$). The other connectives, "and" ($\wedge$), "or" ($\vee$), and "if and only if" ($\leftrightarrow$), can be defined by $\neg$ and $\rightarrow$ as in propositional logic. The operator "possibly" ($\Diamond$) can be defined by $\Diamond A = \neg\Box\neg A$. In addition to the standard rules in propositional logic, K has the following rules:

Necessitation Rule:	If A is a theorem of K, then so is $\Box A$.
Distribution Axiom:	$\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$.

The necessitation rule states that all theorems are necessary and the distribution axiom states that "if it is necessary that if A then B, then if necessarily A then necessarily B". A and B range over all possible formulas for the language.

(M)	$\Box A \rightarrow A$

(4)	$\Box A \rightarrow \Box\Box A$
(5)	$\Diamond A \rightarrow \Diamond\Box A$

(S4):	$\Box\Box\ldots\Box = \Box$ and $\Diamond\Diamond\ldots\Diamond = \Diamond$
(S5):	$00\ldots\Box = \Box$ and $00\ldots\Diamond = \Diamond$, where each 0 is either $\Box$ or $\Diamond$

(B)	$A \rightarrow \Box\Diamond A$

Axiom Name	Axiom	Condition on Frames	R is...
(D)	$\Box A \rightarrow \Diamond A$	$\exists u\ wRu$	Serial
(M)	$\Box A \rightarrow A$	wRw	Reflexive
(4)	$\Box A \rightarrow \Box\Box A$	$(wRv \wedge vRu) \rightarrow wRu$	Transitive
(B)	$A \rightarrow \Box\Diamond A$	$wRv \rightarrow vRw$	Symmetric
(5)	$\Diamond A \rightarrow \Box\Diamond A$	$(wRv \wedge wRu) \rightarrow vRu$	Euclidean
(CD)	$\Diamond A \rightarrow \Box A$	$(wRv \wedge wRu) \rightarrow v = u$	Unique
($\Box$M)	$\Box(\Box A \rightarrow A)$	$wRv \rightarrow vRv$	Shift Reflexive
(C4)	$\Box\Box A \rightarrow \Box A$	$wRv \rightarrow \exists u(wRu \wedge uRv)$	Dense
(C)	$\Diamond\Box A \rightarrow \Box\Diamond A$	$wRv \wedge wRx \rightarrow \exists u(vRu \wedge xRu)$	Convergent

lxi

Temporal Logic

P	"It has at some time been the case that …"
F	"It will at some time be the case that …"
H	"It has always been the case that …"
G	"It will always be the case that …"

Pp ≡ ¬H¬p
Fp ≡ ¬G¬p

Gp→Fp	"What will always be, will be"
G(p→q)→(Gp→Gq)	"If p will always imply q, then if p will always be the case, so will q"
Fp→FFp	"If it will be the case that p, it will be — in between — that it will be"
¬Fp→F¬Fp	"If it will never be that p then it will be that it will never be that p"

p→HFp	"What is, has always been going to be"
p→GPp	"What is, will always have been"
H(p→q)→(Hp→Hq)	"Whatever always follows from what always has been, always has been"
G(p→q)→(Gp→Gq)	"Whatever always follows from what always will be, always will be"

RH: From a proof of p, derive a proof of Hp
RG: From a proof of p, derive a proof of Gp

F∃xp(x)→∃xFp(x)	("If there will be something that is p, then there is now something that will be p")

Spq "q has been true since a time when p was true"
Upq "q will be true until a time when p is true"

$Pp \equiv Sp(p \vee \neg p)$
$Fp \equiv Up(p \vee \neg p)$

$Pp \equiv \exists n(n<0 \ \& \ Fnp)$
$Fp \equiv \exists n(n>0 \ \& \ Fnp)$
$Hp \equiv \forall n(n<0 \rightarrow Fnp)$
$Gp \equiv \forall n(n>0 \rightarrow Fnp)$

$Op \equiv Up(p \& \neg p)$

$Fp \equiv Op \vee OFp$

Pp is true at t if and only if p is true at some time t′ such that t′<t
Fp is true at t if and only if p is true at some time t′ such that t<t′

Hp is true at t if and only if p is true at all times t′ such that t′<t
Gp is true at t if and only if p is true at all times t′ such that t<t′

p is true at all times under all interpretations over any frame in F.
For any frame not in F, there is an interpretation which makes p false at some time.

$Hp \rightarrow Pp$ $\quad \forall t \exists t'(t'<t)$ (unbounded in the past)

Gp→Fp	∀t∃t′(t<t′)	(unbounded in the future)
Fp→FFp	∀t,t′(t<t′ → ∃t″(t<t″<t′))	(dense ordering)
FFp→Fp	∀t,t′(∃t″(t<t″<t′) → t<t′)	(transitive ordering)
FPp → PppFp	∀t,t′,t″((t<t″ & t′<t″) → (t<t′ t=t′ t′<t))	(linear in the past)
PFp → PppFp	∀t,t′,t″((t″<t & t″<t′) → (t<t′ t=t′ t′<t))	(linear in the future)

Kill(Brutus,Caesar,44BC)

Pp ∃t(t<now & p(t))
Fp ∃t(now<t & p(t))
Gp ∀t(t<now → p(t))
Hp ∀t(now<t → p(t))

Holds(Asleep(Mary),(1pm,6pm))
Occurs(Walk-to(John,Station),(1pm,1.15pm))

∀s,i,i′(Holds(s,i) & In(i′,i) → Holds(s,i′))
∀e,i,i′(Occurs(e,i) & In(i′,i) → ¬Occurs(e,i′))

John saw Mary in London on Tuesday.
Therefore, John saw Mary on Tuesday.

∃e(See(John,Mary,e) & Place(e,London) & Time(e,Tuesday)),
Therefore, ∃e(See(John,Mary,e) & Time(e,Tuesday)).

[lxii]

Petri Net

Amdahl's Law

Gene Amdahl, a computer architect, entrepreneur, former IBM employee and one of the creators of the IBM System 360 architecture, devised this method in 1967 to determine the maximum expected improvement to a system when only part of it has been improved. He presented this as an argument against parallel processing. This law is similar to the law of diminished returns, which states that as more input is applied, each additional input unit will produce less additional output. Amdahl's law states that a number of functions or operations must be executed sequentially, decreasing a computer's speed when more processors are added. In other words, the number of tasks that must be completed sequentially limits computational speedup. This causes a bottleneck in the workflow, slowing the overall task. However as the size of a task increases the effect of Amdahl's law decreases. The speedup of a system is:

$$\frac{unimproved\_time}{improved\_time} = speedup = \frac{performance\_with\_improvement}{performance\_without\_improvement}$$

If you make an improvement that greatly increases performance (maybe 100 times or more) in part of a computation but the overall improvement is only 25 percent, then the upper limit for speedup S is:

$$S = \frac{unimproved\_time}{improved\_time} = \frac{1.00}{1.00 - 0.25} = 1.333$$

Note: The unimproved execution time is 1.00 = 100% because this example makes use of the ratio between the two times, not the actual values. Assume that an unimproved computation takes 4 seconds and the improved computation takes 3 seconds. The equation is:

$$S = \frac{unimproved\_time}{improved\_time} = \frac{4\,\text{sec}}{3\,\text{sec}} = 1.333$$

If the improved computation is taken to be 100 percent performance, then by the relationship above the unimproved computation has 75 percent performance with respect to the improved.

$$S = \frac{performance\_with\_improvement}{performance\_without\_improvement} = \frac{100}{75} = 1.333$$

If a computation is improved such that it affects a proportion F_p of the computation, then the improvement will have a speedup S affecting F_p. The improved time for a computation will be equal to the unimproved time multiplied by the sum of the unaffected portion ($1-F_p$) and the speedup reduced affected portion ($F_p \div S$) of the task. To find the improved execution time we use:

$$improved\_time = unimproved\_time \times \left[\left(1 - F_p\right) + \frac{F_p}{S} \right]$$

Continuing the formula above with an affected portion of 40 percent and a speedup of 2.66 times on this portion, we have:

$$improved\_time = 4 \times \left[\left(1 - 0.4\right) + \frac{0.4}{2.66} \right] = 4 \times \left(0.6 \times 0.15\right) = 4 \times 0.75 = 3$$

This method states, assuming that the value for the speed of the unimproved computation is 100 percent, the overall speedup for this computational improvement will be:

$$S = \frac{unimproved\_time}{improved\_time} = \frac{1}{(1 - F_p) + \frac{F_p}{S}}$$

Then plugging in the example proportional values:

$$S = \frac{1}{(1 - 0.4) + \frac{0.4}{2.66}} = \frac{1}{0.75} = 1.33$$

Using time values instead of proportions, we have:

$$S = \frac{4\,\text{sec}}{(4\,\text{sec} - 1.6\,\text{sec}) + \frac{1.6\,\text{sec}}{2.66}} = \frac{4\,\text{sec}}{3\,\text{sec}} = 1.33$$

Amdahl's law for parallelization states that the sequential fraction F_s of a task that cannot be performed in parallel and the fraction $F_p = (1\text{-}F_s)$ that can gives the following formula for maximum speedup by N_p processors:

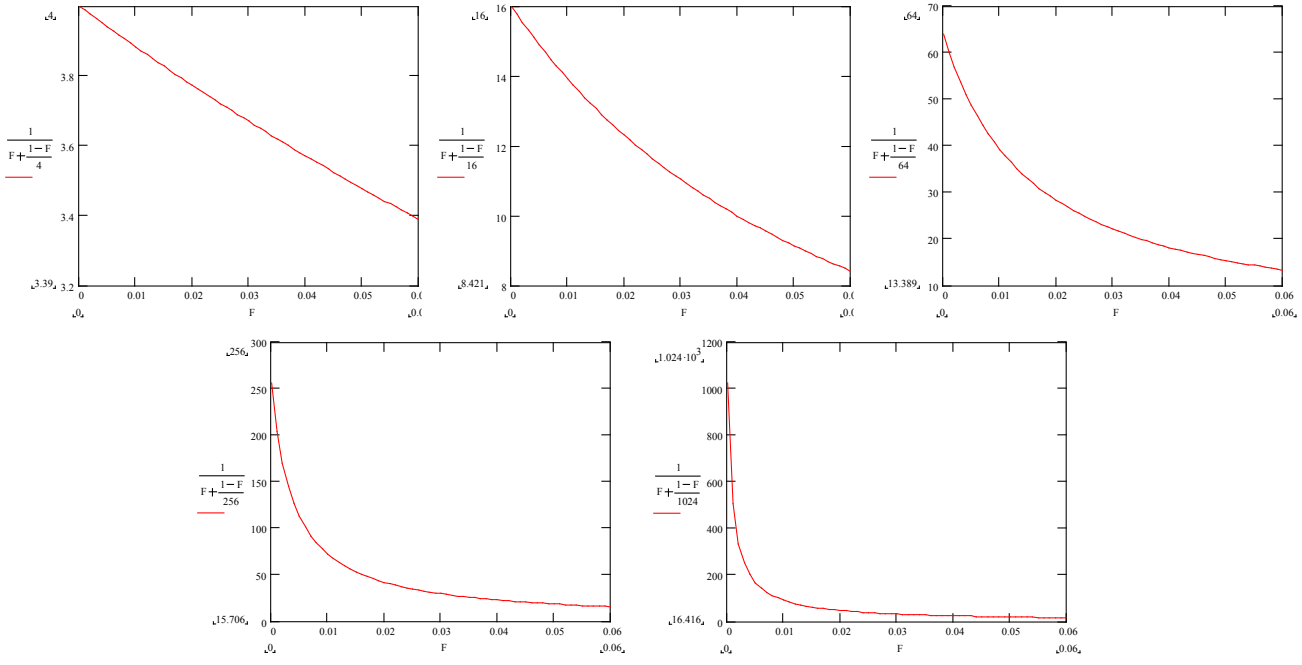
$$S = \frac{1}{F_s + \frac{1 - F_s}{N_p}}$$

As N approaches infinity, the maximal speedup approaches $1/F_s$. As the $(1\text{-}F_s)/N_p$ value becomes very small, the price paid for marginal performance increases. Assume that $F_s = 0.06$. Then $F_p = 1\text{-}F_s = 0.94$. For 4 processors:

$$S = \frac{1}{0.06 + \frac{1 - 0.06}{4}} = \frac{1}{0.06 + \frac{0.94}{4}} = \frac{1}{0.06 + 0.235} = \frac{1}{0.295} = 3.3898$$

The table below shows the run time, speedup, efficiency and cost for processors $N_p=\{1,2,4,\ldots,1024\}$, where $F_s = 0.06$ and $F_p = 0.94$. Notice that the speedup per additional processor is much less as N_p increases, causing greater cost and less efficiency. The graphs show the effect on speedup (y-axis) with respect to F_s (x-axis) with increasing N_p.

Processors(N_p)	1	2	4	8	16	32	64	128	256	512	1024
Run Time	1024.00	542.72	302.08	181.76	121.60	91.52	76.48	68.96	65.20	63.32	62.38
Speedup	1.0000	1.8868	3.3898	5.6338	8.4211	11.1888	13.3891	14.8492	15.7055	16.1718	16.4155
Efficiency	100.00%	94.34%	84.75%	70.42%	52.63%	34.97%	20.92%	11.60%	6.13%	3.16%	1.60%
Cost	1.00	1.06	1.18	1.42	1.90	2.86	4.78	8.62	16.30	31.66	62.38



The graphs have values N_p of 4, 16, 64, 256 and 1024. Notice that as the value N_p increases, the area under the curve decreases, meaning that the non-parallizable part of the serial program has a greater effect and the degeneration occurs faster as N_p increases.

Amdahl's intention was to show *"the continued validity of the single processor approach and of the weaknesses of the multiple processor approach"*. His paper proposed arguments to support his proposal, such as:

- *"The nature of this overhead appears to be sequential so that it is unlikely to be amenable to parallel processing techniques."*
- *"A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel performance rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."*

Gustafson's Law

In 1988, John L. Gustafson proposed the notion that massively parallel processing was beneficial because Amdahl's law implies that the parallel part of the computation and the number of processors is independent [[lxiii]]. He proposed a formula for a scaled speedup based on an observation that in most real world computations *"the problem size scales with the number of processors"*. His proposed formula is:

$$S = \frac{fraction\_serial + (fraction\_parallel \times number\_of\_processors)}{(fraction\_serial + fraction\_parallel = 1)}$$

$$= \frac{F_s + (1 - F_s) \times N_p}{F_s + (1 - F_s)}$$

$$= \frac{F_s + (1 - F_s) \times N_p}{1}$$

$$= F_s + N_p - N_p F_s$$

$$= N_p + (F_s - N_p F_s)$$

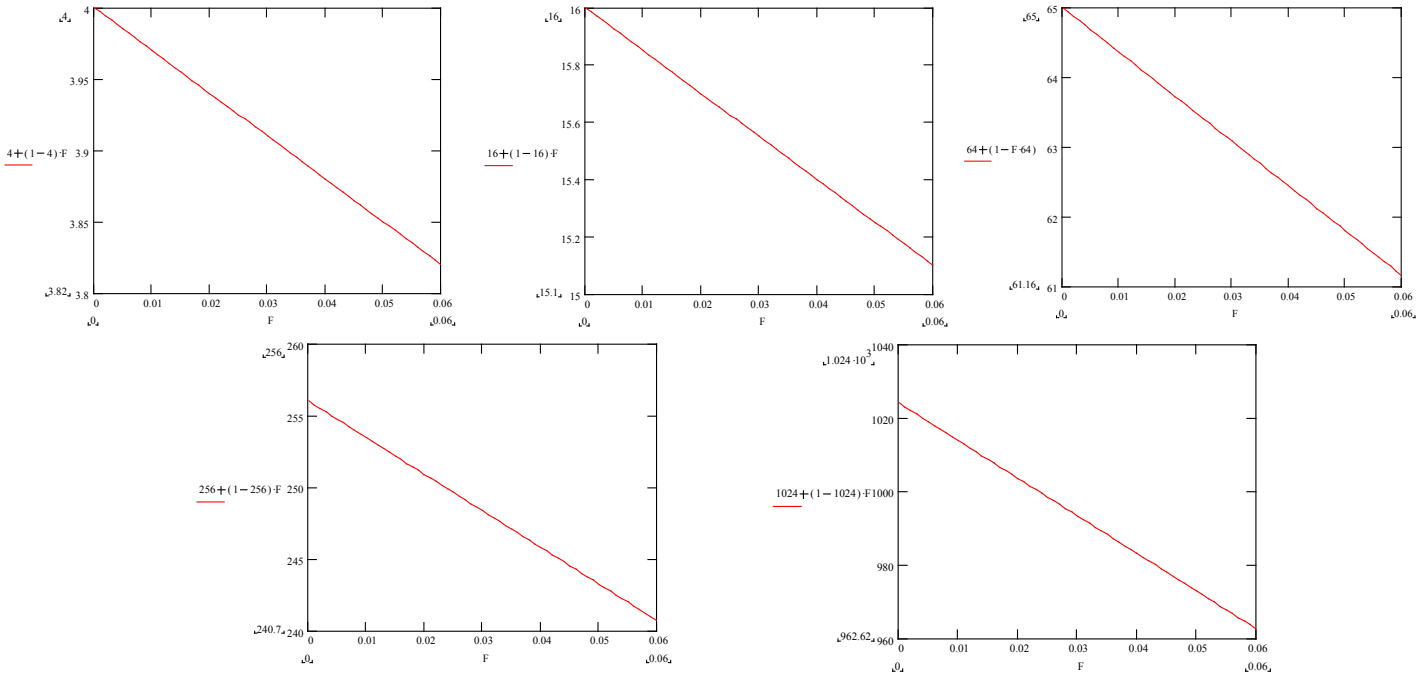
$$= N_p + (1 - N_p) \times F_s$$

where S is the speedup, the serial portion is F_s and N_p is the number of processors. Again, assume that $F_s = 0.06$. Then $F_p = 1–F_s = 0.94$. For 4 processors:

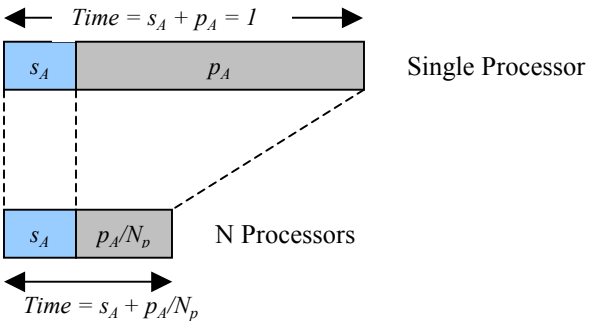
$$S = N_p + (1 - N_p) \times F_s = 4 + (1 - 4) \times 0.06 = 4 - 0.18 = 3.82$$

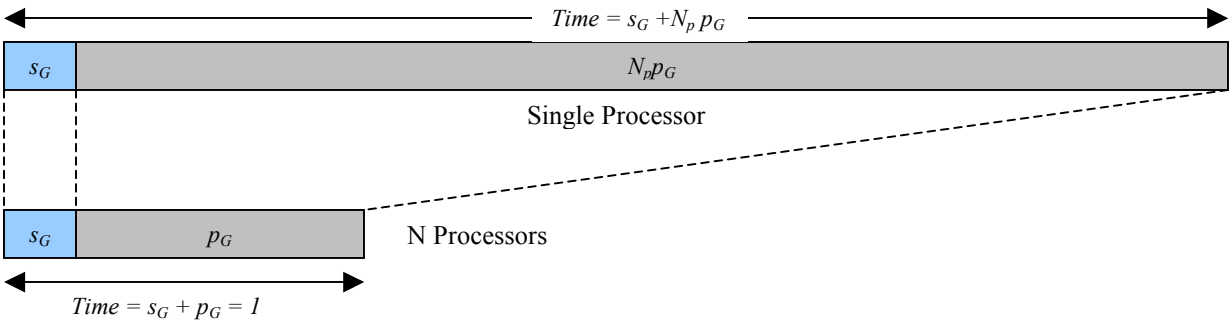
The table and graphs below show the same data as in Amdahl but using Gustafson's law.

Processors(N)	1	2	4	8	16	32	64	128	256	512	1024
Run Time	1024.0000	527.8351	268.0628	135.0923	67.8146	33.9748	17.0043	8.5064	4.2543	2.1274	1.0638
Speedup	1.0000	1.9400	3.8200	7.5800	15.1000	30.1400	60.2200	120.3800	240.7000	481.3400	962.6200
Efficiency	100.00%	97.00%	95.50%	94.75%	94.38%	94.19%	94.09%	94.05%	94.02%	94.01%	94.01%
Cost	1.0000	1.0309	1.0471	1.0554	1.0596	1.0617	1.0628	1.0633	1.0636	1.0637	1.0638



Consider the following diagrams, which are similar to those in Gustafson's paper:





Under Gustafson's proposal, increasing the number of processors has little affect on cost or efficiency and an almost linear speedup, as shown in the graphs above. The problem with this method of evaluating computational speedup is that the serial and parallel programs perform different numbers of operations on the primary task because the task for the parallel implementation is N_p times larger than that of the serial. If the parallelized operation were matrix multiplication on n^2 matrices for $n_s = 10$, there would be $10^3 = 1000$ multiplication and 1000 addition operations in the serial program. If you scale up the problem for $N_p = 4$ processors the multiplication operations must increase to 4000 and the matrix n_p size must increase to:

$$\sqrt[3]{4000} = \sqrt[3]{1000} \times \sqrt[3]{4} = 10 \times 1.5874 \approx 16$$

Because matrix multiplication is $O(n^3)$ complexity, increasing the size of the matrix, even minimally, creates a much bigger job. An observation by Yuan Shi was proposed in [lxiv], where an equivalence between Amdahl's Law and Gustafson's Law is explained. The relationship is based on the adjustment to the serial fraction in Amdahl's Law, call it F_{sA}, and the unadjusted serial fraction used in Gustafson's Law, call it F_{sG}, such that:

$$F_{sA} = \frac{1}{1 + \frac{(1 - F_{sG}) \times N_p}{F_{sG}}}$$

As an example, consider a task that has serial fraction $F_{sG} = 0.05$ with 1024 processors. Amdahl's Law would predict speedup S to be:

$$S = \frac{1}{F_{sG} + \frac{1 - F_{sG}}{N_p}} = \frac{1}{0.05 + \frac{1 - 0.05}{1024}} = \frac{1}{0.05 + \frac{0.95}{1024}}$$

$$= \frac{1}{0.05 + 0.0009277} = \frac{1}{0.0509277} = 19.635666$$

Gustafson's Law predicts:

$$S = N_p + (1 - N_p) \times F_s = 1024 + (1 - 1024) \times 0.05$$

$$= 1024 - 1023 \times 0.05 = 1024 - 51.15 = 972.85$$

However when the serial fraction F_{sA} is calculated from F_{sG} using the equation above, we have:

$$F_{sA} = \frac{1}{1 + \frac{(1 - F_{sG}) \times N_p}{F_{sG}}} = \frac{1}{1 + \frac{(1 - 0.05) \times 1024}{0.05}} = \frac{1}{1 + \frac{972.8}{0.05}}$$

$$= \frac{1}{1 + 19456} = \frac{1}{19457} = 5.14\text{E-}05$$

We substitute F_{sA} for F_{sG} and solve:

$$S = \frac{1}{F_s + \frac{1 - F_s}{N_p}} = \frac{1}{5.14\text{E-}05 + \frac{1 - 5.14\text{E-}05}{1024}} = \frac{1}{5.14\text{E-}05 + \frac{0.99994}{1024}}$$

$$= \frac{1}{5.14\text{E-}05 + 9.7650\text{E-}04} = \frac{1}{1.0279\text{E-}03} = 972.85$$

For this situation, the claim of equivalent results with Gustafson's Law by obtaining F_{sA} from F_{sG}, as defined above, and substituting F_{sA} for F_{sG} in Amdahl's Law is true. The table below shows that this is true for all number of processors, where Np = {1, 2, 4, 8, …, 1024} and $F_{sG} = 0.05$.

Processors N_p	1	2	4	8	16	32	64	128	256	512	1024
F_{sG}	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
F_{sA}	0.05	0.025641	0.012987	0.0065359	0.0032787	0.001642	0.0008217	0.000411	0.0002055	0.0001028	5.14E-05
Amdahl-F_{sG}	1	1.9047619	3.4782609	5.9259259	9.1428571	12.54902	15.421687	17.414966	18.618182	19.284369	19.635666
Gustafson	1	1.95	3.85	7.65	15.25	30.45	60.85	121.65	243.25	486.45	972.85
Amdahl-F_{sA}	1	1.95	3.85	7.65	15.25	30.45	60.85	121.65	243.25	486.45	972.85

The table below shows that this is also true for all F_{sG}, where F_{sG} = {0.01, 0.02, …, 0.90, 0.1, 0.2} and Np = 1024.

Processors N_p	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
F_{sG}	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1	0.2
F_{sG}	9.864E-06	1.993E-05	3.02E-05	4.069E-05	5.14E-05	6.233E-05	7.35E-05	8.491E-05	9.657E-05	0.0001085	0.0002441
Amdahl-F_{sG}	91.184328	47.716682	32.313033	24.427481	19.635666	16.415518	14.102741	12.361178	11.002471	9.9128751	4.9805447
Gustafson	1013.77	1003.54	993.31	983.08	972.85	962.62	952.39	942.16	931.93	921.7	819.4
Amdahl-F_{sA}	1013.77	1003.54	993.31	983.08	972.85	962.62	952.39	942.16	931.93	921.7	819.4

Performance Metrics

Performance metrics are basically measures of computer and/or network system behavior over a given period of time. The four primary types of performance metrics:

- Latency
- Throughput
- Efficiency
- Availability
- Reliability
- Utilization

Latency is also called response time. It is a measure of the delay between the initial time of a request for some service and the time that the service arrives, expressed in units of elapsed time. The elapsed time between the completion of dialing a phone number and the first ring, the time that a router holds a packet, and the time spent waiting for a Web page to be displayed after a hyperlink is clicked are all latency metrics. It can be stated as a statistical distribution. An example is a server that must acknowledge 99.9% of client requests in one second or less.

Throughput, also called capacity, is the rate that results arrive or the amount of work done in a given time. It is measured in the quantity of units per time. Megabits per second of data transmitted across a network, transactions completed per minute in a transaction server, and gigabytes of data per second transferred across a system buss are all throughput metrics. The theoretical maximum throughput is called bandwidth. The bandwidth of a 400Mhz, 64-bit data bus is 25.6Gb/s (400Mhz × 64-bit) but the actual throughput is less because of padding between data blocks and control protocols.

The ratio of usable throughput compared to the bandwidth is called efficiency. The efficiency of a 400Mhz, 64-bit data bus, with a throughput of 20.48Gb/s, is 80% (20.48Gb/s ÷ 25.6Gb/s). Goodput is the arrival rate of good data packets across a computer network. If, on average, 920 packets arrive uncorrupted at the destination, the goodput is said to be 92%.

Availability is the percentage of time that a system is available to provide service. If a server is down for 15 minutes each day for maintenance, it has 98.96% availability (1425min ÷ 1440min).

The reliability metric reports the mean time between failures (MTBF), which indicates the average period that the system is usable. The mean time to repair (MTTR) is the average time to recover from failures.

Utilization is the percentage of time that a component in the system is active. Utilization is typically measured as a percentage. The capacity or maximum throughput of a system is reached when the utilization of the busiest component is 100%. Many systems have a utilization threshold because as utilization approaches 100%, system latency quickly increases.

Performance metrics for parallel systems include the following:

- Runtime
- Speedup
- Efficiency
- Cost
- Scalability

The run time of a parallel system is elapsed time from the instance of execution of the master or controller program until the last program in the parallel system terminates. T_s usually denotes the serial or single processor run time of a task is and T_p usually denotes the parallel run time.

Speedup, usually denoted by S, is the ratio calculated by dividing the serial run time of a particular task by the parallel run time for the same task:

$$S = \frac{T_s}{T_p}$$

As an example, if two size n matrices are to be multiplied, the operation has complexity $\Theta(n^3)$. Assuming that the run time for the operation a single processor is n^3, the theoretical speedup, ignoring parallel system overhead, for 2 processors is:

$$T_1 = n^3, T_2 = \frac{n^3}{2}, S = \frac{T_1}{T_2} = \frac{n^3}{n^3/2} = 2$$

Be careful not to make the following mistake for parallel time and speedup:

$$T_s = n^3, T_p = \left(\frac{n}{2}\right)^3 = \frac{n^3}{8}, S = \frac{T_s}{T_p} = \frac{n^3}{n^3/8} = 8$$

This assumes a change in the overall problem size, which is false because matrix multiplication is n^3 multiplications and n^3 additions, regardless of how many processors are used.

Efficiency, usually denoted as E, is the ratio calculated by dividing the speedup S by the number of processors N_p, which measures the percentage of time that a processor is working on the primary task. For the matrix multiplication example the efficiency is:

$$E = \frac{S}{N_p} = \frac{2}{2} = 1 = 100\%$$

Parallel system overhead T_o can decrease system efficiency. Parallel system overhead consists of all the necessary operations to manage and setup the parallel system, divide the task among the processors, transmit the task to the worker processes, collect the results from the processes and compile the results. It may include pieces of the sequential program that cannot be parallelized $T_{1\text{-}p}$. Hence a more realistic formula for the run time with n processors T_n , where T_c is the time spent on computation of the task, is:

$$T_n = T_c + T_o + T_{1-p}$$

Assume that the following values are valid for the matrix multiplication above:

- Sequential run time T_1 120sec
- Parallel computation time T_c 60sec
- Parallel overhead T_o 20sec
- Assume no non-parallizable code T_{1-p} 0sec

Then speedup would be

$$T_1 = 60\,\text{sec}, T_2 = T_c + T_o + T_{1-p} = 60\,\text{sec} + 20\,\text{sec} + 0\,\text{sec} = 80\,\text{sec}, S = \frac{120\,\text{sec}}{80\,\text{sec}} = 1.5 = 150\%$$

This is somewhat less than the previous speedup.

The cost C of a parallel system is calculated by multiplying the parallel run T_n time and the number of processors N_p divided by the sequential run time T_1:

$$C = \frac{T_n \times N_p}{T_1}$$

The values in the example above, ignoring overhead, would be.

$$C = \frac{T_n \times N_p}{T_1} = \frac{60\,\text{sec} \times 2}{120\,\text{sec}} = \frac{120\,\text{sec}}{120\,\text{sec}} = 1$$

This equation is shows that the parallel system is optimal because the increase in speed is proportional with the number of processors added. Typically costs are not optimal. Considering the overhead in the example above, we have:

$$C = \frac{T_n \times N_p}{T_1} = \frac{80\,\text{sec} \times 2}{120\,\text{sec}} = \frac{160\,\text{sec}}{120\,\text{sec}} = 1.333\,.$$

Timing Models

Gathering System Performance Data

Gathering Network Performance Data

Optimal Load balancing

Load balancing is the efficient distribution of the workload over all available processors, keeping all processors busy until the task is complete. Not all machines will have the same computational capacity. Some machines may have lower processor speeds or other tasks that consume system resources. The idea is to shift more work to processors that can accommodate it. Optimization is the modification of a system to improve performance and efficiency. Optimal load balancing occurs when the latency of requests is minimized, computation is distributed equally across all processors, system throughput is maximized, and the system completes all tasks in the least possible time. An absolutely optimal system is rare and can be difficult to produce. Optimization usually involves compromise. Performance or efficiency in one part of a system may have to be sacrificed to optimize another part.

Successful optimization requires the development of sound algorithms and a functional prototype. Challenges to load balancing include problems with timing, communication, synchronization, and iterative tasks and branching that may depend conditions elsewhere in the parallel system. If tasks in a parallel system have differing execution times, one or more processors will have to wait for the longest executing task to finish. Communication and synchronization will occur over some communication channel, such as the system buss or a network. Systems that require an abundance of communication may cause a bottleneck in these channels. If the channel is shared between multiple processes, competition for the resource may cause contention in heavily loaded channels. Loops and branches can easily lead to non-deterministic program behavior if measures are not employed to prevent it.

There are two classifications of load balancing: static and dynamic. Static load balancing uses statistics, based on the ability of each processor's ability to perform, to share the burden of the workload. Dynamic load balancing shares work by dynamically averaging job size based on the performance of participating processors. Dynamic load balancing requires more communication synchronization between processes, which consumes communication time. However, the tradeoff is that dynamic load balancing can handle unexpected delays when jobs take unreasonable amounts of time, where static load balancing cannot. If a task is taking longer than anticipated, some work can be sent to other processes. The extra communication may decrease throughput but the processes will be kept busy. It is also important to mention that load balancing should reduce the

overall run time for the system. If it takes less time to complete the task without it, we should forgo load balancing.[lxv]

About Synergy

Blue text: Copied and pasted from Getting Started by Dr. Shi
Red text: Copied and pasted from syng_man.ps by Dr. Shi

Introduction to The Synergy Project

What is Synergy?

Synergy is a parallel computing system using a Stateless Parallel Processing (SPP) principle. It is a simplified prototype implementation of a Stateless Machine (SLM). It lacks backbone fault tolerance and stateful process fault tolerance. It is also known to have an inefficient tuple matching engine in comparison to the full implementation of SLM.

SPP is based on coarse-grain dataflow processing. A full SLM implementation will offer, in addition to all benefits that Synergy affords, a more efficient tuple matching engine and a non-stop computing platform with total fault tolerance for stateful processes and for the backbone. An SLM can be considered a higher form of Symmetric MultiProcessor (SMP).

Functionally, Synergy can be thought of as an equivalent to PVM, Linda or MPI/MPICH.

Synergy uses passive objects for inter-process(or) communication. It offers programming ease, load balancing and fault tolerance benefits. The application-programming interface (API) is a small set of operators defined on the supported object types, such as tuple space, file and database. Synergy programs use a conventional open-manipulate-close sequence for each passive object. Each Synergy program is individually compiled using a conventional compiler and a Synergy Language Injection Library (LIL). A parallel application is synthesized through a configuration specification (CSL) and an automatic processor-binding algorithm. Synergy runtime system can execute multiple parallel applications on the same cluster at the same time.

Synergy API blends well into the conventional sequential programs. It is particularly helpful for reengineering legacy applications. It even allows parallel processing of mixed PVM and MPI programs.

Synergy and SPP

Synergy is a prototype implementation of a StateLess Machine (SLM). It uses a Passive Object-Flow Programming (POFP) method to offer programming ease, process fault Tolerance and high efficiency using cluster of networked computers.

In principle, a Stateless Parallel Processing (SPP) system requires total location transparency for all processes (running programs). This affords three important non-functional features: ease of programming, fault tolerance and load balancing.

In programming, this means that location (host address and port) dependent IPC primitives are NOT allowed. Consequently, a special asynchronous IPC layer (of Passive Objects) is used for inter-process communication and synchronization. The SPP runtime system can automatically determine the optimal process-to-processor binding during the execution of a parallel application. This additional IPC layer does carry some overheads in comparison to direct IPC systems such as MPI/PVM. In return, it gives three critical benefits: programming ease, load balancing and fault tolerance support at the architecture level.

Why Synergy?

First, one hidden fact that has not been mentioned in any high performance multiprocessor's literature is that the use of multiple processors for a single application necessarily reduces its availability if any processor failure can halt the entire application. The current state of art in parallel processing is still under the shadow of this gloomy fact. SPP offers an approach that promises breakthroughs in both high performance and high availability using multi-processors. Synergy is the first prototype designed to explore architectural flaws and to validate the claims of SPP.

Second, technically, separation of functional programming from process coordination and resource management functions can ease parallel programming while maintaining high performance and availability. Although many believe that explicit manipulation of processes and data objects can produce highly optimized parallel codes, we believe ease of programming, high performance and high availability are of a higher importance in making industrial strength parallel applications using multiprocessors.

Synergy Philosophy

Facilitating the best use of computing and networking resources for each application is the key philosophy in Synergy. We advocate competitive resource sharing as opposed to ``cycle stealing." The tactic is to reduce processing time for each application. Multiple running applications would fully exploit system resources. The realization of the objectives, however, requires both quantitative analysis and highly efficient tools.

It is inevitable that parallel programming and debugging will be more time consuming than single thread processing regardless how well the application programming interface (API) is designed. The illusive parallel processing results taught us that we must have quantitatively convincing reasons to processing an application in parallel before committing to the potential expenses (programming, debugging and future maintenance.)

We use Timing Models to evaluate the potential speedups of a parallel program using different processors and networking devices [13]. Timing models capture the orders of timing costs for computing, communication, disk I/O and synchronization requirements. We can quantitatively examine an application's speedup potential under various processor and networking assumptions. The analysis results delineate the limit of hopes. When applied to practice, timing models provide guidelines for processing grain selection and experiment design.

Efficiency analysis showed that effective parallel processing should follow an incremental coarse-to-fine grain refinement method. Processors can be added only if there are unexplored parallelism, processors are available and the network is capable of carrying the anticipated load. Hard-wiring programs to processors will only be efficient for a few special applications with restricted input at the expense of programming difficulties.

To improve performance, we took an application-oriented approach in the tool design. Unlike conventional compilers and operating systems projects, we build tools to customize a given processing environment for a given application. This customization defines a new infrastructure among the pertinent compilers, operating systems and the application for effective resource exploitation. Simultaneous execution of multiple parallel applications permits exploiting available resources for all users. This makes the networked processors a fairly real ``virtual supercomputer."

An important advantage of the Synergy compiler-operating system-application infrastructure is the higher level portability over existing systems. It allows written parallel programs to adapt into any programming, processor and networking technologies without compromising performance.

An important lesson we learned was that mixing parallel processing, resource management and functional programming tools in one language made tool automation and parallel programming unnecessarily difficult. This is especially true for parallel processors employing high performance uni-processors.

Building timing models before parallel programming can determine the worthiness of the undertaking in the target multiprocessor environment and prevent costly design mistakes. The analysis can also provide guidelines for parallelism grain size selection and experiment design (http://joda.cis.temple.edu/~shi/super96/timing/timing.html)

Except for server programs, all parallel processing applications can be represented by a coarse grain dataflow graph (CGDG). In CGDG, each node is either a repetition node or a non-repetition node. A repetition node contains either an iterative or recursive process. The edges represent data dependencies. It should be fairly obvious that CGDG must be acyclic.

CGDG fully exhibits potential effective (coarse grain) parallelism for a given application. For example, the SIMD parallelism is only possible for a repetition node. The MIMD parallelism is possible for any 1-K branch in CGDG. Pipelines exist along all sequentially dependent paths provided that there are repetitive input data feeds. The actual processor assignment determines the deliverable parallelism.

Any repetition node can be processed in a coarse grain SIMD (or scatter-and-gather) fashion. The implementation of a repetition node is to have a master and a worker program connected via two tuple space objects. The master is responsible for distributing the work tuples and collecting results. The worker is responsible for computing the results from a given input and delivering the results.

For all other components in the graph, one can use tuple space or pipe. The use of file and database (yet to be implemented) objects is defined by the application.

Following the above description results in a static IPC graph using passive objects. The programmer's job is to compose parallel programs communicating with these objects.

History

Synergy V3.0 is an enhancement to Synergy V2.0 (released in early 1994). Earlier versions of the same system appeared in the literature under the names of MT (1989), ZEUS (1986), Configurator (1982) and Synergy V1.0 (1992) respectively.

Major Components and Inner Workings of Synergy

Technically, the Synergy system is an automatic client/server software generation system that can form an effective parallel processor for each application using multiple distributed Unix or Linux computers. This parallel processor is specifically engineered to process programs inter-connected in an application dependent IPC (Inter-Program Communication/ Synchronization) graph using industry standard compilers, operating systems and communication protocols. This IPC graph exhibits application dependent coarse grain SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data) and pipeline parallelisms.

Synergy V3.0 supports three passive data objects for program-to-program communication and synchronization:

1. Tuple space (a FIFO ordered tuple data manager)
2. Pipe (a generic location independent indirect message queue)
3. File (a location transparent sequential file)

A passive object is any structured data repository permitting no object creation functions. All commonly known large data objects, such as databases, knowledge bases, hashed files, and ISAM files, can be passive objects provided the object creating operators are absent. Passive objects confine dynamic dataflows into a static IPC graph for any parallel application. This is the basis for automatic customization.

POFP uses a simple open-manipulate-close sequence for each passive object. An one-dimensional Coarse-To-Fine (CTF) decomposition method (see Adaptable Parallel Application Development section for details) can produce designs of modular parallel programs using passive objects. A global view of the connected parallel programs reveals application dependent coarse grain SIMD, MIMD and pipeline potentials. Processing grain adjustments are done via the work distribution programs (usually called Masters). These adjustments can be made without changing codes. All parallel programs can be developed and compiled independently.

What are in Synergy? (Synergy Kernel with Explanation)

The first important ingredient in Synergy is the confinement of inter-program communication and synchronization (IPC) mechanisms. They convert dynamic

application dataflows to a static, bipartite IPC graph. In Synergy, this graph is used to automate process coordination and resource management. In other words, Synergy V3.0 uses this static IPC graph to automatically map parallel programs onto set of networked computers that forms a virtual multiprocessor. In the full SLM implementation, this static IPC graph will be implemented via a self-healing backbone.

Synergy v3.0 contains the following service components:

- A language injection library (LIL). This is the API programmers use to compose parallel programs. It contains operators defined on supported passive objects, such as tuple space, file, pipe or database.
- Two memory resident service daemons (PMD and CID). These daemons resolve network references and are responsible for remote process/object execution and management.
- Two dynamic object daemons (TSH and FAH). These daemons are launched before every parallel application begins and are removed after the application terminates. They implement the defined semantics of LIL operators.
- A customized Distributed Application Controller (DAC). This program actually synthesizes a multiprocessor application. It conducts processor binding and records relevant information about all processes involved in the application until completion. DAC represents a customized virtual multiprocessor for each application.
- Synergy shell: (prun and pcheck). These programs are Synergy runtime user interface.
    - prun launches a parallel application
    - pcheck is a runtime monitor for managing multiple parallel applications and processes

ADD PRUN AND LIL INFO HERE

Program ``pcheck" functions analogously as the ``ps" command in Unix. It monitors parallel applications and keeps track of parallel processes of each application. Pcheck also allows killing running processes or applications if necessary.

To make remote processors listening to personal commands, there are two light weight utility daemons: the Command Interpreter Daemon (cid) and the Port Mapper Daemon (pmd). Cid interprets a limited set of process control commands from the network for each user account. In other words, parallel users on the same processor need different cid's. Pmd (the peer leader) provides a "yellow page" service for locating local cid's. Pmd is automatically started by any cid and is transparent to all users.

FDD is a Fault Detection Daemon. It is activated by an option in the prun command to detect worker process failures at runtime.

Synergy V3.0 requires no root privileged processes. All parallel processes assume respective user security and resource restrictions defined at account creation. Parallel use of multiple computers imposes no additional security threat to the existing systems. Theoretically, there should be one object daemon for each supported object type. For the three supported types: tuple space, pipe and files, we saved the pipe daemon by implementing it directly in LIL. Thus, Synergy V3.0 has only two object daemons: the Tuple Space Handler (tsh) and the File Access Handler (fah). The object daemons, when activated, talk to parallel programs via the LIL operators under the user defined identity (via CSL). They are potentially resource hungry. However they only "live" on the computers where they are needed and permitted.

Optimal processor assignment is theoretically complex. Synergy's automatic processor binding algorithm is extremely simple: unless specifically designated, it binds all tuple space objects, one master and one worker to a single processor. Other processors run the worker-type (with repeatable logic) processes. Since network is the bottleneck, this binding algorithm minimizes network traffic thus promising good performance for most applications using the current tuple matching engine. The full implementation of SLM will have a distributed tuple matching engine that promises to fulfill a wider range of performance requirements.

Fault tolerance is a natural benefit of the SPP design. Processor failures discovered before a run are automatically isolated. Worker processor failures during a parallel execution is treated in V3.0 by a "tuple shadowing" technique. Synergy V3.0 can automatically recover the lost data from a lost worker with little overhead. This feature brings the availability of a multiprocessor application to be equal to that of a single processor and is completely transparent to application programs.

Synergy provides the basis for automatic load balancing. However, optimal load balancing requires adjusting tuple sizes. Tuple size adjustments can adapt guided self-scheduling [1], factoring [2] or fixed chunking using the theory of optimal granule size for load balancing [3].

Synergy V3.0 runs on clusters of workstations. This evaluation copy allows unlimited processors across multiple file systems (*requires one binary installation per file system).

Comparisons with Other Systems

Synergy vs. PVM/MPI

PVM/MPI is a direct message passing system [5,6] that requires inter-process communication be carried out based on process task id's. This requirement forces an extra user-programming layer if fault tolerance and load balancing are desired. This is because for load balancing and fault tolerance, working data cannot be "hard wired" to specific processors. An "anonymous" data item can only be supplied using an additional data management layer providing a tuple space-like interface. In this sense, we consider PVM/MPI a lower level parallel API as compared to Linda and Synergy.

Fault tolerant and load balanced parallel programs typically require more inter-process communication than direct message passing since they refresh their states frequently in order to expose more “stateless moments” – critical to load balance and fault tolerance. This is a tradeoff that users must make before adapting the Synergy parallel programming platform.

Synergy vs. Linda

The original Linda implementation [4] uses a virtual global tuple space implemented using a compile time analysis method. The main advantage of the Linda method is the potential to reduce communication overhead. It was believed that many tuple access patterns could be un-raveled into single lines of communication. Thus the compiler can build the machine dependent codes directly without going through an intermediate runtime daemon that would potentially double the communication latency of each tuple transmission. However, experiments indicate that majority applications do not have static tuple access patterns that a compiler can easily discern. As a result, increased communication overhead is inevitable.

The compile time tuple binding method is also detrimental to fault tolerance and load balancing.

Another problem in the Linda design is the limited scalability. Composing all parallel programs in one file and compiled by a single compiler makes programming unnecessarily complex and is impractical to large-scale applications. It also presents difficulties for mixed language processing.

In comparison, Synergy uses dynamic tuple binding at the expense of increased communication overhead by using dynamic tuple space daemons. In the full SLM implementation, this overhead will be reduced by a distributed tuple matching engine. Practical computational experiments indicate that synchronization overhead (due to load imbalance) logged more time than communication. Thus Synergy's load balancing advantage can be used to offset its increased communication overhead.

Parallel Programming and Processing in Synergy

A parallel programmer must use the passive objects for communication and synchronization purposes. These operations are provided via the language injection library (LIL). LIL is linked to source programs at compilation time to generate hostless binaries that can run on any binary compatible platforms.

After making the parallel binaries the interconnection of parallel programs (IPC graph) should be specified in CSL (Configuration Specification Language). Program ``prun'' starts a parallel application. Prun calls CONF to process the IPC graph and to complete the program/object-to-processor assignments automatically or as specified. It then activates DAC to start appropriate object daemons and remote processes (via remote cid's). It preserves the process dependencies until all processes are terminated.

Building parallel applications using Synergy requires the following steps:

1. Parallel program definitions. This requires, preferably, establishing timing models for a given application. Timing model analysis provides decomposition guidelines. Parallel programs and passive objects are defined using these guidelines.
2. Individual program composition using passive objects.
3. Individual program compilation. This makes hostless binaries by compiling the source programs with the Synergy object library (LIL). It may also include moving the binaries to the $HOME/bin directory when appropriate.
4. Application synthesis. This requires a specification of program-to-program communication and synchronization graph (in CSL). When needed, user preferred program-to-processor bindings are to be specified as well.
5. Run (prun). At this time the program synthesis information is mapped on to a selected processor pool. Dynamic IPC patterns are generated (by CONF) to guide the behavior of remote processes (via DAC and remote cid's). Object daemons are started and remote processes are activated (via DAC and remote cid's).
6. Monitor and control (pcheck).

Load Balancing and Performance Optimization

Fault Tolerance

Installing and Configuring Synergy

Red text: Copied and pasted from syng_man.ps by Dr. Shi
Gray text: Copied and pasted from a document by Dr. Shi

Basic Requirements

In addition to installing Synergy V3.0 on each computer cluster, there are four requirements for each ``parallel'' account:

1. An active SNG_PATH symbol definition pointing to the directory where Synergy V3.0 is installed. It is usually /usr/local/synergy.
2. An active command search path ($SNG_PATH/bin) pointing to the directory holding the Synergy binaries.
3. A local host file ($HOME/.sng_hosts). Note that this file is only necessary for a host to be used as an application submission console.
4. An active personal command interpreter (cid) running in the background. Note that the destination of future parallel process's graphic display should be defined before starting cid.

Since the local host file is used each time an application is started, it needs to reflect a) all accessible processors; and b) selected hosts for the current application.

Unpacking

To uncompress, at Unix prompt, type

```
% uncompress synergy-3.0.tar.Z
```

To untar,

```
% tar -xvf synergy-3.0.tar
```

A directory called "synergy" will be created and all files unpacked under this directory.

Compiling

To compile, change to the synergy directory and type

```
% make
```

The current version has been tested on these platforms:

- SUN 3/4, SunOs
- IBM RS6000, AIX
- DEC Alpha, OSF/1
- DEC ULTRIX
- Silicon Graphics, SGI
- HP, HP-UX
- CDC cyber, EP/IX

The makefile will try to detect the operating system and build binaries, libraries and sample applications. You may need to edit the makefile if your system requires special flags, and/or if your include/library path is nonstandard. Check the makefile for detail.

Configuring the Synergy Environment

After the installation procedure is complete, some minor changes must be made to the computers environment to access the Synergy system. When using a UNIX/Linux system we enter commands in a command-line environment called a shell. This shell must be configured to recognize the Synergy system. The two most used shells are C Shell (csh) and Bourne Again Shell (bash). Examples of configuration or profile files will be shown below for csh and bash. Because these files are hidden, you must type:

```
ls –a
```

and press the enter key at the terminal command prompt to view them.

To configure csh, you must edit the ".cshrc" file in your home directory by adding the line:

```
setenv SNG_PATH synergy_directory
```

where ***synergy_directory*** is the directory containing all the binary files and the Synergy object library. Next, add the Synergy binary directory to the path definition by typing:

```
set path=($SNG_PATH/bin $path)
```

at the command line and pressing enter. It is important to add `$SNG_PATH/bin` before `$path`, since "prun" may be overloaded in some operating systems (such as SunOS 5.9). To activate the new settings enter:

```
source .cshrc
```

at the command prompt.

An example of a ".cshrc" file after the settings have been changed, with the changes in bold, for the SunOS is:

```
#ident "@(#)local.cshrc    1.2    00/05/01 SMI"
umask 077
set path=( /usr/users/shi/synergy/bin /opt/SUNWspro/bin /bin /usr/bin /usr/ucb
/etc ~ )
if ( -d ~/bin ) then
  set path=( $path ~/bin )
endif
set path=( $path . )

if ( $?prompt ) then
       set history=32
endif

set prompt="[%n@%m %c ]%#"

# Initialize new variables
setenv LD_LIBRARY_PATH ""
setenv MANPATH "/opt/SUNWspro/man"

# Adding the SUN Companion CD Software, including GCC 2.95
set path=( $path /opt/sfw/bin /opt/sfw/sparc-sun-solaris2.9/bin /usr/local/bin
)
setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:/opt/sfw/lib:/usr/local/lib"
setenv MANPATH "/opt/sfw/man:/usr/local/man:${MANPATH}"

# Adding Usr-Local-Bin
set path=( $path /usr/local/bin )
setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:/usr/local/lib"
setenv MANPATH "/usr/local/man:${MANPATH}"

# Usr-Sfw
set path=( $path /usr/sfw/bin )
setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:/usr/lib:/usr/sfw/lib"
setenv MANPATH "${MANPATH}:/usr/man:/usr/sfw/man"

# DT Window Manager
set path=( $path /usr/dt/bin )
#setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/dt/lib
setenv MANPATH "${MANPATH}:/usr/dt/man"

# GNOME
```

```
set path=( $path /usr/share/gnome )
setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:/usr/share/lib"
setenv MANPATH "${MANPATH}:/usr/share/man"
setenv SNG_PATH /usr/users/shi/synergy

# SBIN
set path=( $path /sbin /usr/sbin )
```

An example ".cshrc" file for Linux OS would be:

```
set path = ( ~ ~/bin /usr/java/j2sdk_nb/j2sdk1.4.2/bin $path \
       /usr/local/X11R6/bin /usr/local/bin /usr/bin /usr/users/shi/synergy/bin
. )

set noclobber
limit coredumpsize 0

#         aliases for all shells

#alias cd            'cd \!*;set prompt="`hostname`:`pwd`>"'
alias pwd           'echo $cwd'
alias edt          'textedit -fn screen.b.14'

set history = 1000
set savehist = 400
set ignoreeof
set prompt="%m:%~>"

alias help          man
alias key           'man -k'

setenv EDITOR 'pico -t'
setenv MANPATH /usr/man:/usr/local/man:/usr/share/man
setenv WWW_HOME http://www.cis.temple.edu
setenv NNTPSERVER netnews.temple.edu
setenv SNG_PATH /usr/users/shi/synergy
#source ~/.aliases

# auto goto client
[ "$tty" != "" ] && [ `hostname` = 'lucas' ] && exec gotoclient
```

To configure bash you must edit the ".bash_profile" file by adding the lines:

```
SNG_PATH = synergy_directory

export SNG_PATH
```

where ***synergy_directory*** is the directory containing all the binary files and the Synergy object library and add the following entry to the path:

```
/usr/users/shi/synergy/bin:
```

To activate the new settings enter:

```
source .bash_profile
```

at the command prompt.

Below is an example of the ".bash_profile" file for the Linux OS.

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=/usr/users/shi/synergy/bin:/usr/java/j2sdk_nb/j2sdk1.4.2/bin:$PATH:$HOME/bin
```

SNG_PATH = usr/users/shi/synergy

```
export PATH
export SNG_PATH
unset USERNAME

# auto goto client
[ "$TERM" != "dumb" ] && [ `hostname` = 'lucas' ] && exec gotoclient
```

Activating a Processor Pool

To activate your personal parallel processors, you will need to start one "cid" one each of the host either manually or by some shell script at least once.

In addition, if you have special remote display requirements, you need to setup your display characteristics BEFORE starting cid. For example you may want to monitor a simulator running on many hosts and "steer" the program as it goes.

In this case, you will need to open as many windows as the number of hosts you want to monitor and telnet (rlogin) to these hosts. Then you need to start a cid in each of these hosts after you designate your display host. Cid has memories. It will send the local display to the designated host as by the "setenv DISPLAY" command.

To start cid enter:

```
%cid &
```

Cid will try to connect to another daemon named "pmd". If it could not contact the peer leader in three times, it will start the peer leader automatically.

To check for the total processor accessibility at any host, enter:

```
%cds
```

This command checks host status for all SELECTED entries in your host file.

Note that you DO NOT have to re-start cid on the de-selected host if you want to re-select them if a cid is already running, unless you want to change the display setup.

Using Synergy

The Synergy System

Using Synergy's Tuple Space Objects

Using Synergy's Pipe Objects

Using Synergy's File Objects

Compiling Synergy Applications

Running Synergy Applications

Debugging Synergy Applications

Tuple Space Object Programming

A Simple Application – Hello Synergy!

The first example given in most introductory computer programming books is the "Hello World!" program. To get started with Synergy programming, the "Hello Synergy!" program will be the first example. The master program (tupleHello1Master.c) simply opens a tuple space, puts the message in the tuple space and terminates. The worker programs (tupleHello1Worker.c) open the tuple space, read the message from the tuple space, display the message and terminate. The following example programs can be found in the example01 directory.

The following is the tuple space "Hello Synergy!" master program:

```
#include <stdio.h>
#include <sys/resource.h>

main(){
  int tplength;       // Length of ts entry
  int status;         // Return status for tuple operations
  int P;              // Number of processors
  int tsd;            // Problem tuple space identifier
  char host[128];     // Host machine name
  char tpname[20];    // Identifier of ts entry

  // Message sent to workers
  char sendMsg[50] = "Hello Synergy!\0";

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Master: Opening tuple space\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  printf("Master: Tuple space open complete\n");

  // Get number of processors
  P = cnf_getP();
  printf("Master: Processors %d\n", P);

  // Send 'Hello Synergy!' to problem tuple space
  // Set length of send entry
  tplength = sizeof(sendMsg);
  // Set name of entry to host
  strcpy(tpname, host);
  printf("Master: Putting '%s' Length %d Name %s\n",
          sendMsg, tplength, tpname);
  // Put entry in tuple space
```

```
  status = cnf_tsput(tsd, tpname, sendMsg, tplength);
  printf("Master: Put '%s' complete\n", sendMsg);
  // Sleep 1 second
  sleep(1);

  // Terminate program
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space "Hello Synergy!" worker program:

```
#include <stdio.h>
#include <sys/resource.h>

main(){
  int tsd;           // Problem tuple space identifier
  int status;        // Return status for tuple operations
  int tplength;      // Length of ts entry
  char host[128];    // Host machine name
  char tpname[20];   // Identifier of ts entry
  char recdMsg[50];  // Message received from master

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple space
  printf("Worker: Opening tuple space\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  printf("Worker: Tuple space open complete\n");

  // Set name to any
  strcpy(tpname,"*");
  // Read problem from problen tuple space
  tplength = cnf_tsread(tsd, tpname, recdMsg, 0);
  printf("Worker: Taking item (%s)\n", tpname);

  // Normal receive
  if (tplength > 0){
    printf("Worker: Took message: %s from %s\n",
            recdMsg, tpname);
  }

  // Terminate program
  printf("Worker: Terminated\n");
  cnf_term();
}
```

Before the master and worker programs can execute these programs, a Command Specification Language (csl) file must be created. It would be much more convenient to use a makefile to compile the programs. Examples of both are below.

The csl file the programs is:

```
configuration: tupleHello1;

m: master = tupleHello1Master
       (factor = 1
       threshold = 1
       debug = 0
        )
   -> f: problem
        (type = TS)
   -> m: worker = tupleHello1Worker
       (type = slave)
   -> f: result
       (type = TS)
   -> m: master;
```

The makefile for the programs is:

```
CFLAGS = -O1
OBJS = -L$(SNG_PATH)/obj -lsng -lnsl -lsocket

all : nxdr copy

nxdr :  master1 worker1

master1 : tupleHello1Master.c
       gcc $(CFLAGS) -o tupleHello1Master tupleHello1Master.c $(OBJS)

worker1 : tupleHello1Worker.c
       gcc $(CFLAGS) -o tupleHello1Worker tupleHello1Worker.c $(OBJS)

copy : tupleHello1Master tupleHello1Worker
       cp tupleHello1Master $(HOME)/bin
       cp tupleHello1Worker $(HOME)/bin
```

To run the “Hello Synergy!” distributed application:

1. Make the executables by typing “make” and pressing the enter key.
2. Run the application by typing “prun tupleHello1” and pressing the enter key.

The screen output for the master terminal should resemble:

```
[c615111@owin ~/fpc01 ]>prun tupleHello1
== Checking Processor Pool:
++ Benchmark (186) ++ (owin) ready.
== Done.
== Parallel Application Console: (owin)
== CONFiguring: (tupleHello1.csl)
== Default directory: (/usr/classes/cis6151/c615111/fpc01)
++      Automatic program assignment: (worker)->(owin)
```

```
++      Automatic program assignment: (master)->(owin)
++      Automatic object assignment: (problem)->(owin) pred(1) succ(1)
++      Automatic object assignment: (result)->(owin) pred(1) succ(1)
== Done.
== Starting Distributed Application Controller ...
Verifying process [|(c615111)|*/tupleHello1Master
CID verify ****'d process (bin/tupleHello1Master)
Verifying process [|(c615111)|*/tupleHello1Worker
CID verify ****'d process (bin/tupleHello1Worker)
** (tupleHello1.prcd) verified, all components executable.
CID starting object (result)
CID starting object (problem)
CID starting program. path (bin/tupleHello1Master)
Master: Opening tuple space
CID starting program. path (bin/tupleHello1Worker)
Master: Tuple space open complete
Master: Processors 1
Master: Putting 'Hello Synergy!' Length 50 Name owin
Master: Put 'Hello Synergy!' complete
Worker: Opening tuple space
** (tupleHello1.prcd) started.
Worker: Tuple space open complete
Worker: Taking item (owin)
Worker: Took message: Hello Synergy! from owin
Worker: Terminated
CID. subp(27144) terminated
Setup exit status for (27144)
Master: Terminated
CID. subp(27143) terminated
Setup exit status for (27143)
CID. subp(27141) terminated
Setup exit status for (27141)
== (tupleHello1) completed. Elapsed [1] Seconds.
CID. subp(27142) terminated
Setup exit status for (27142)
[c615111@owin ~/fpc01 ]>
```

The output for the worker terminal should resemble:

```
CID verify ****'d process (bin/tupleHello1Worker)
CID starting program. path (bin/tupleHello1Worker)
Worker: Opening tuple space
Worker: Tuple space open complete
Worker: Taking item (owin)
Worker: Took message: Hello Synergy! from owin
Worker: Terminated
CID. subp(21015) terminated
Setup exit status for (21015)
```

The output shows Synergy's distributed application initialization screen output, the execution screen output of the master and worker programs, and termination screen output of both programs and the distributed application.

Sending and Receiving Data

Hello Workers!—Hello Master!!!

In this example application, the master (tupleHello2Master.c) sends the message "Hello Workers!" to all workers (tupleHello2Worker.c) and gets the response "Hello Master!!!" and the worker's name from each worker. The source code, makefile and csl file for this application is located in the example02 directory.

The following is the tuple space "Hello Workers!—Hello Master!!!" master program:

```
#include <stdio.h>
#include <sys/resource.h>

main() {
  int tplength;       // Length of ts entry
  int status;         // Return status for tuple operations
  int P;              // Number of processors
  int i;              // Counter index
  int res;            // Result tuple space identifier
  int tsd;            // Problem tuple space identifier
  char host[128];     // Host machine name
  char tpname[20];    // Identifier of ts entry
  char recdMsg[50];   // Message received from workers

  // Message sent to workers
  char sendMsg[50] = "Hello Workers!\0";

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Master: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  // Open result tuple space
  res = cnf_open("result",0);
  printf("Master: Tuple spaces open complete\n");

  // Get number of processors
  P = cnf_getP();
  printf("Master: Processors %d\n", P);

  // Send 'Hello Synergy!' to problem tuple space
  // Set length of send entry
  tplength = sizeof(sendMsg);
  // Set name of entry to host
  strcpy(tpname, host);
  printf("Master: Putting '%s' Length %d Name %s\n",
          sendMsg, tplength, tpname);
  // Put entry in tuple space
```

```
  status = cnf_tsput(tsd, tpname, sendMsg, tplength);
  printf("Master: Put '%s' complete\n", sendMsg);
  // Sleep 1 second
  sleep(1);

  // Receive 'Hello Back!!!' from result tuple space
  for(i=0; i<P; i++){
    printf("Master: Waiting for reply\n");
    // Set name to any
    strcpy(tpname,"*");
    // Get result from result tuple space
    tplength = cnf_tsget(res, tpname, recdMsg, 0);
    printf("Master: Taking item from %s\n", tpname);
    printf("Master: Took message '%s'\n", recdMsg);
  }

  // Terminate program
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space “Hello Workers!—Hello Master!!!” worker program:

```
#include <stdio.h>
#include <sys/resource.h>

main(){
  int tsd;           // Problem tuple space identifier
  int res;           // Result tuple space identifier
  int status;        // Return status for tuple operations
  int tplength;      // Length of ts entry
  char host[128];    // Host machine name
  char tpname[20];   // Identifier of ts entry
  char recdMsg[50];  // Message received from master

  // Message sent back to master
  char sendMsg[50] = "Hello Master!!!\0";

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Worker: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  // Open result tuple space
  res = cnf_open("result",0);
  printf("Worker: Tuple spaces open complete\n");

  // Set name to any
  strcpy(tpname,"*");
  // Read problem from problen tuple space
  tplength = cnf_tsread(tsd, tpname, recdMsg, 0);
  printf("Worker: Taking item (%s)\n", tpname);
```

```
  // Normal receive
  if (tplength > 0){
    printf("Worker: Took message: %s from %s\n",
            recdMsg, tpname);
    // Set size of entry
    tplength = sizeof(sendMsg);
    // Set name to host
    sprintf(tpname,"%s", host);
    printf("Worker: Put '%s' Length %d Name %s\n",
            sendMsg, tplength, tpname);
    // Put response in result tuple space
    status = cnf_tsput(res, tpname, sendMsg, tplength);
    printf("Worker: Reply sent\n");
  }

  // Terminate program
  printf("Worker: Terminated\n");
  cnf_term();
}
```

The makefile and csl file are similar to the "Hello Synergy!" program except that all occurrences of "tupleHello1…" is changed to "tupleHello2…" in both files. To run the "Hello Synergy!" distributed application:

1. Make the executables by typing "make" and pressing the enter key.
2. Run the application by typing "prun tupleHello2" and pressing the enter key.

The screen output for the master terminal with Synergy's initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc02 ]>prun tupleHello2
Master: Tuple spaces open complete
Master: Processors 2
Master: Putting 'Hello Workers!' Length 50 Name owin
Master: Put 'Hello Workers!' complete
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Taking item owin
Worker: Took message: 'Hello Workers!' from owin
Worker: Put 'Hello Master!!!' Length 50 Name owin
Worker: Reply sent
Worker: Terminated
Master: Waiting for reply
Master: Taking item from saber
Master: Took message 'Hello Master!!!'
Master: Waiting for reply
Master: Taking item from owin
Master: Took message 'Hello Master!!!'
Master: Terminated
[c615111@owin ~/fpc02 ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Taking item owin
Worker: Took message: 'Hello Workers!' from owin
Worker: Put 'Hello Master!!!' Length 50 Name saber
Worker: Reply sent
Worker: Terminated
```

Sending and Receiving Data Types

Sending Various Data Types

Synergy can put and get more than characters from its tuple space. The following example shows how to put various data types into a tuple space and get various data types out of a tuple space. The master program (tuplePassMaster.c) puts different data types into the problem tuple space, and the worker (tuplePassWorker.c) gets them, displays them and puts messages in the result tuple space identifying which data types it took. This application also uses a distributed semaphore to ensure that the workers take data properly. It also demonstrates the difference between the cnf_read() and cnf_get() functions. The tuplePass application is located in the example03 directory. The tuplePass.h file has the definitions for the constant and the data structure used in the application.

The following is the tuple space "data type passing" master program:

```
#include <stdio.h>
#include <sys/resource.h>

#include "tuplePass.h"

main(){
  int tplength;       // Length of ts entry
  int status;         // Return status for tuple operations
  int P;              // Number of processors
  int i;              // Counter index
  int res;            // Result tuple space identifier
  int tsd;            // Problem tuple space identifier
  int sem;            // Semaphore
  char host[128];     // Host machine name
  char tpname[20];    // Identifier of ts entry
  char recdMsg[50];   // Message received from workers

  // Different datatypes to send to workers
  // Integer sent to worker
  int num = 12000;
  int *numPtr = &num;
  // Long integer sent to worker
  long lnum = 1000000;
  long *lnumPtr = &lnum;
  // Float sent to worker
  float frac = 0.5;
  float *fracPtr = &frac;
  // Double sent to worker
  double dfrac = 12345.678;
  double *dfracPtr = &dfrac;
  // Integer array sent to worker
  int numArr[MAX] = {0,1,2,3,4};
```

```
// Double array sent to worker
double dblArr[MAX] = {10000.1234, 2000.567,
                      300.89, 40.0, 5.01};
// String sent to worker
char sendMsg[50] = "A text string.\0";
// Struct sent to worker
struct person bob = {"Bob",
                     "123 Broad St.",
                     "Pliladelphia", "PA", "19124",
                      20, "brown", 70.5, "red"};

// Get host machine name
gethostname(host, sizeof(host));

// Open tuple spaces
printf("Master: Opening tuple spaces\n");
// Open problem tuple space
tsd = cnf_open("problem",0);
// Open result tuple space
res = cnf_open("result",0);
printf("Master: Tuple spaces open complete\n");

// Get number of processors
P = cnf_getP();
printf("Master: Processors %d\n", P);

// Put semaphore in problem tuple space
// Set name to sem
strcpy(tpname,"sem");
// Set length for semaphore
tplength = sizeof(int);
// Place the semaphore signal in problem ts
printf("Master: Putting semaphore\n");
status = cnf_tsput(tsd, tpname, &sem, tplength);

// Put int num in ts
// Set length of send entry
tplength = sizeof(int);
// Set name of entry to num
strcpy(tpname, "D_num");
printf("Master: Putting '%d' Length %d Name %s\n",
        num, tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, numPtr, tplength);
printf("Master: Put '%d' complete\n", num);

// Put long lnum in ts
// Set length of send entry
tplength = sizeof(long);
// Set name of entry to lnum
strcpy(tpname, "D_lnum");
printf("Master: Putting '%ld' Length %d Name %s\n",
        lnum, tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, lnumPtr, tplength);
printf("Master: Put '%ld' complete\n", lnum);
```

```
// Put float frac in ts
// Set length of send entry
tplength = sizeof(float);
// Set name of entry to frac
strcpy(tpname, "D_frac");
printf("Master: Putting '%f' Length %d Name %s\n",
        frac, tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, fracPtr, tplength);
printf("Master: Put '%f' complete\n", frac);

// Put double dfrac in ts
// Set length of send entry
tplength = sizeof(double);
// Set name of entry to dfrac
strcpy(tpname, "D_dfrac");
printf("Master: Putting '%g' Length %d Name %s\n",
        dfrac, tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, (char *)dfracPtr, tplength);
printf("Master: Put '%g' complete\n", dfrac);

// Put int array numArr in ts
// Set length of send entry
tplength = sizeof(int)*MAX;
// Set name of entry to numArr
strcpy(tpname, "D_numArr");
printf("Master: Putting\n  ");
for(i=0; i<MAX; i++)
  printf("%d ", numArr[i]);
printf("\n  Length %d Name %s\n", tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, (char *)numArr, tplength);
printf("Master: Put '%s' complete\n", tpname);

// Put int array dblArr in ts
// Set length of send entry
tplength = sizeof(double)*MAX;
// Set name of entry to dblArr
strcpy(tpname, "D_dblArr");
printf("Master: Putting\n  ");
for(i=0; i<MAX; i++)
  printf("%g ", dblArr[i]);
printf("  \nLength %d Name %s\n", tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, (char *)dblArr, tplength);
printf("Master: Put '%s' complete\n", tpname);

// Put struct bob in ts
// Set length of send entry
tplength = sizeof(struct person);
// Set name of entry to bob
strcpy(tpname, "D_bob");
printf("Master: Putting\n");
printf("  %s\n", bob.name);
```

```
  printf("  %s %s, %s %s\n",
         bob.address, bob.city, bob.state, bob.zip);
  printf("  %d %s %f %s\n",
         bob.age, bob.eyes, bob.height, bob.hair);
  printf("  Length %d Name %s\n", tplength, tpname);
  // Put entry in tuple space
  status = cnf_tsput(tsd, tpname, bob, tplength);
  printf("Master: Put struct bob complete\n");

  // Put string in ts
  // Set length of send entry
  tplength = sizeof(sendMsg);
  // Set name of entry to msg
  strcpy(tpname, "D_msg");
  printf("Master: Putting '%s' Length %d Name %s\n",
         sendMsg, tplength, tpname);
  // Put entry in tuple space
  status = cnf_tsput(tsd, tpname, sendMsg, tplength);
  printf("Master: Put '%s' complete\n", sendMsg);

  // Receive results from result tuple space
  for(i=0; i<8; i++){
    printf("Master: Waiting for reply\n");
    // Set name to any
    strcpy(tpname,"*");
    // Get result from result tuple space
    tplength = cnf_tsget(res, tpname, recdMsg, 0);
    printf("Master: Taking item from (%s)\n", tpname);
    printf("Master: %s took '%s'\n", tpname, recdMsg);
  }

  // Send terminal signal to workers
  printf("Master: Putting terminal signal in problem ts\n");
  // Set length of send entry
  tplength = sizeof(int);
  // Set name of entry to term
  strcpy(tpname, "D_term");
  // Put entries in tuple space
  status = cnf_tsput(tsd, tpname, numPtr, tplength);
  printf("Master: Put terminal in ts\n");

  // Terminate program
  printf("Master: Terminated\n");
  cnf_term();

}
```

The following is the tuple space data type passing worker program:

```
#include <stdio.h>
#include <sys/resource.h>

#include "tuplePass.h"

main(){
```

```
int tsd;           // Problem tuple space identifier
int res;           // Result tuple space identifier
int status;        // Return status for tuple operations
int tplength;      // Length of ts entry
int i;             // Counter index
int sem = 0;       // Semaphore
char host[128];    // Host machine name
char tpname[20];   // Identifier of ts entry
char sendMsg[50];  // Message sent back to master

// Different datatypes to receive from master
// Integer received from master
int num;
// Long integer received from master
long lnum;
// Float received from master
float frac;
// Double received from master
double dfrac;
// Integer array received from master
int numArr[MAX];
// Double array received from master
double dblArr[MAX];
// String received from master
char recdMsg[50];
// Struct received from master
struct person bob;

// Initialize sendMsg
strcpy(sendMsg, "");

// Get host machine name
gethostname(host, sizeof(host));

// Open tuple spaces
printf("Worker: Opening tuple spaces\n");
// Open problem tuple space
tsd = cnf_open("problem",0);
// Open result tuple space
res = cnf_open("result",0);
printf("Worker: Tuple spaces open complete\n");

while(1){
  // Set name to sem
  strcpy(tpname,"sem");
  // Read semaphore from problem tuple space
  tplength = cnf_tsget(tsd, tpname, &sem, 0);
  printf("Worker: Taking semaphore\n");
  // Set name to any
  strcpy(tpname,"D_*");
  tplength = cnf_tsread(tsd, tpname, recdMsg, 0);
  printf("Worker: Taking item %s\n", tpname);


  // Get int num from ts
  if(!strcmp(tpname, "D_num")){
```

```
      // Read problem from problem tuple space
      tplength = cnf_tsget(tsd, tpname, &num, 0);
      // Record the data type received
      strcpy(sendMsg, tpname);
      // Display the data
      printf("Worker: took %s '%d'\n", tpname, num);
      // Send reply back to master
      // Set size of entry
      tplength = sizeof(sendMsg);
      // Set name to host
      strcpy(tpname, host);
      printf("Worker: Put '%s' Length %d Name %s\n",
             sendMsg, tplength, tpname);
      // Put response in result tuple space
      status = cnf_tsput(res, tpname, sendMsg, tplength);
      printf("Worker: Reply sent\n");
    }

    // Get int lnum from ts
    else if(!strcmp(tpname, "D_lnum")){
      // Read problem from problem tuple space
      tplength = cnf_tsget(tsd, tpname, &lnum, 0);
      // Record the data type recieve
      strcpy(sendMsg, tpname);
      // Display the data
      printf("Worker: took  %s '%ld'\n", tpname, lnum);
      // Send reply back to master
      // Set size of entry
      tplength = sizeof(sendMsg);
      // Set name to host
      strcpy(tpname, host);
      printf("Worker: Put '%s' Length %d Name %s\n",
              sendMsg, tplength, tpname);
      // Put response in result tuple space
      status = cnf_tsput(res, tpname, sendMsg, tplength);
      printf("Worker: Reply sent\n");
    }

    // Get int frac from ts
    else if(!strcmp(tpname, "D_frac")){
      // Read problem from problem tuple space
      tplength = cnf_tsget(tsd, tpname, &frac, 0);
      // Record the data type received
      strcpy(sendMsg, tpname);
      // Display the data
      printf("Worker: took %s '%f'\n", tpname, frac);
      // Send reply back to master
      // Set size of entry
      tplength = sizeof(sendMsg);
      // Set name to host
      strcpy(tpname, host);
      printf("Worker: Put '%s' Length %d Name %s\n",
             sendMsg, tplength, tpname);
      // Put response in result tuple space
      status = cnf_tsput(res, tpname, sendMsg, tplength);
      printf("Worker: Reply sent\n");
```

```
  }

  // Get double dfrac from ts
  else if(!strcmp(tpname, "D_dfrac")){
    // Read problem from problem tuple space
    tplength = cnf_tsget(tsd, tpname, &dfrac, 0);
    // Record the data type received
    strcpy(sendMsg, tpname);
    // Display the data
    printf("Worker: took (%s) '%g'\n", tpname, dfrac);
    // Send reply back to master
    // Set size of entry
    tplength = sizeof(sendMsg);
    // Set name to host
    strcpy(tpname, host);
    printf("Worker: Put '%s' Length %d Name %s\n",
           sendMsg, tplength, tpname);
    // Put response in result tuple space
    status = cnf_tsput(res, tpname, sendMsg, tplength);
    printf("Worker: Reply sent\n");
  }

  // Get integer array numArr
  else if(!strcmp(tpname, "D_numArr")){
    // Read problem from problem tuple space
    tplength = cnf_tsget(tsd, tpname, numArr, 0);
    // Record the data type received
    strcpy(sendMsg, tpname);
    // Display the data
    printf("Worker: took %s\n  ", tpname);
    for(i=0; i<MAX; i++)
      printf("%d ", numArr[i]);
    printf("\n  Length(%d) Name(%s)\n", tplength, tpname);
    // Send reply back to master
    // Set size of entry
    tplength = sizeof(sendMsg);
    // Set name to host
    strcpy(tpname, host);
    printf("Worker: Put '%s' Length %d Name %s\n",
           sendMsg, tplength, tpname);
    // Put response in result tuple space
    status = cnf_tsput(res, tpname, sendMsg, tplength);
    printf("Worker: Reply sent\n");
  }

  // Get double array dblArr
  else if(!strcmp(tpname, "D_dblArr")){
    // Read problem from problem tuple space
    tplength = cnf_tsget(tsd, tpname, dblArr, 0);
    // Record the data type received
    strcpy(sendMsg, tpname);
    // Display the data
    printf("Worker: took %s\n  ", tpname);
    for(i=0; i<MAX; i++)
      printf("%g ", dblArr[i]);
    printf("\n  Length %d Name %s\n", tplength, tpname);
```

```
      // Send reply back to master
      // Set size of entry
      tplength = sizeof(sendMsg);
      // Set name to host
      strcpy(tpname, host);
      printf("Worker: Put '%s' Length %d Name %s\n",
              sendMsg, tplength, tpname);
      // Put response in result tuple space
      status = cnf_tsput(res, tpname, sendMsg, tplength);
      printf("Worker: Reply sent\n");
    }

    // Get struct person bob
    else if(!strcmp(tpname, "D_bob")){
      // Read problem from problem tuple space
      tplength = cnf_tsget(tsd, tpname, &bob, 0);
      // Record the data type received
      strcpy(sendMsg, tpname);
      // Display the data
      printf("Worker: took\n");
      printf("  %s\n", bob.name);
      printf("  %s %s, %s %s\n", bob.address,
             bob.city, bob.state, bob.zip);
      printf("  %d %s %f %s\n", bob.age, bob.eyes,
             bob.height, bob.hair);
      printf("  Length %d Name %s\n", tplength, tpname);
      // Send reply back to master
      // Set size of entry
      tplength = sizeof(sendMsg);
      // Set name to host
      strcpy(tpname, host);
      printf("Worker: Put '%s' Length %d Name %s\n",
             sendMsg, tplength, tpname);
      // Put response in result tuple space
      status = cnf_tsput(res, tpname, sendMsg, tplength);
      printf("Worker: Reply sent\n");
    }

    // Get string
    else if(!strcmp(tpname, "D_msg")){
      // Read problem from problem tuple space
      tplength = cnf_tsget(tsd, tpname, recdMsg, 0);
      // Record the data type received
      strcpy(sendMsg, tpname);
      // Display the data
      printf("Worker: took %s '%s'\n", tpname, recdMsg);
      // Send reply back to master
      // Set size of entry
      tplength = sizeof(sendMsg);
      // Set name to host
      strcpy(tpname, host);
      printf("Worker: Put '%s' Length %d Name %s\n",
             sendMsg, tplength, tpname);
      // Put response in result tuple space
      status = cnf_tsput(res, tpname, sendMsg, tplength);
      printf("Worker: Reply sent\n");
```

```
    }

    // Get terminal
    else if(!strcmp(tpname, "D_term")){
      printf("Worker: Received terminal\n");
      // Set name to sem
      strcpy(tpname,"sem");
      // Set length for semaphore
      tplength = sizeof(int);
      // Replace the semaphore signal in problem ts
      printf("Worker: Putting semaphore\n");
      status = cnf_tsput(tsd, tpname, &sem, tplength);
      break;
    }

    // Set name to sem
    strcpy(tpname,"sem");
    // Set length for semaphore
    tplength = sizeof(int);
    // Replace the semaphore signal in problem ts
    printf("Worker: Putting semaphore\n");
    status = cnf_tsput(tsd, tpname, &sem, tplength);
    // Sleep 1 second
    sleep(1);

  }

  // Terminate program
  printf("Worker: Terminated\n");
  cnf_term();

}
```

The makefile and csl file are similar to the last two applications except in the naming of the application objects and files. To run the data passing distributed application:

1. Make the executables by typing “make” and pressing the enter key.
2. Run the application by typing “prun tuplePass” and pressing the enter key.

The screen output for the master terminal with Synergy’s initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc03 ]>prun tuplePass2
Master: Opening tuple spaces
Master: Tuple spaces open complete
Master: Processors 2
Master: Putting semaphore
Master: Putting '12000' Length 4 Name D_num
Master: Put '12000' complete
Master: Putting '1000000' Length 4 Name D_lnum
Master: Put '1000000' complete
Master: Putting '0.500000' Length 4 Name D_frac
```

```
Master: Put '0.500000' complete
Master: Putting '12345.7' Length 8 Name D_dfrac
Master: Put '12345.7' complete
Master: Putting
  0 1 2 3 4
Length 20 Name D_numArr
Master: Put 'D_numArr' complete
Master: Putting
  10000.1 2000.57 300.89 40 5.01
  Length 40 Name D_dblArr
Master: Put 'D_dblArr' complete
Master: Putting
  Bob
  123 Broad St. Pliladelphia, PA 19124
  20 brown 70.500000 red
  Length 164 Name D_bob
Master: Put struct bob complete
Master: Putting 'A text string.' Length 50 Name D_msg
Master: Put 'A text string.' complete
Master: Waiting for reply
Master: Taking item from saber
Master: saber took 'D_num'
Master: Waiting for reply
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Taking semaphore
Worker: Taking item D_lnum
Worker: took  D_lnum '1000000'
Worker: Put 'D_lnum' Length 50 Name owin
Master: Taking item from owin
Master: owin took 'D_lnum'
Master: Waiting for reply
Worker: Reply sent
Worker: Putting semaphore
Master: Taking item from saber
Master: saber took 'D_frac'
Master: Waiting for reply
Worker: Taking semaphore
Worker: Taking item D_dfrac
Worker: took (D_dfrac) '12345.7'
Worker: Put 'D_dfrac' Length 50 Name owin
Master: Taking item from owin
Master: owin took 'D_dfrac'
Master: Waiting for reply
Worker: Reply sent
Worker: Putting semaphore
Master: Taking item from saber
Master: saber took 'D_numArr'
Master: Waiting for reply
Worker: Taking semaphore
Worker: Taking item D_dblArr
Worker: took D_dblArr
  10000.1 2000.57 300.89 40 5.01
  Length 40 Name D_dblArr
Worker: Put 'D_dblArr' Length 50 Name owin
Worker: Reply sent
```

```
Worker: Putting semaphore
Master: Taking item from owin
Master: owin took 'D_dblArr'
Master: Waiting for reply
Master: Taking item from saber
Master: saber took 'D_bob'
Master: Waiting for reply
Worker: Taking semaphore
Worker: Taking item D_msg
Worker: took D_msg 'A text string.'
Worker: Put 'D_msg' Length 50 Name owin
Worker: Reply sent
Worker: Putting semaphore
Master: Taking item from owin
Master: owin took 'D_msg'
Master: Putting terminal signal in problem ts
Master: Put terminal in ts
Master: Terminated
Worker: Taking semaphore
Worker: Taking item D_term
Worker: Received terminal
Worker: Putting semaphore
Worker: Terminated
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Taking semaphore
Worker: Taking item D_num
Worker: took D_num '12000'
Worker: Put 'D_num' Length 50 Name saber
Worker: Reply sent
Worker: Putting semaphore
Worker: Taking semaphore
Worker: Taking item D_frac
Worker: took D_frac '0.500000'
Worker: Put 'D_frac' Length 50 Name saber
Worker: Reply sent
Worker: Putting semaphore
Worker: Taking semaphore
Worker: Taking item D_numArr
Worker: took D_numArr
  0 1 2 3 4
  Length(20) Name(D_numArr)
Worker: Put 'D_numArr' Length 50 Name saber
Worker: Reply sent
Worker: Putting semaphore
Worker: Taking semaphore
Worker: Taking item D_bob
Worker: took
  Bob
  123 Broad St. Philadelphia, PA 19124
```

```
  20 brown 70.500000 red
  Length 164 Name D_bob
Worker: Put 'D_bob' Length 50 Name saber
Worker: Reply sent
Worker: Putting semaphore
Worker: Taking semaphore
Worker: Taking item D_term
Worker: Received terminal
Worker: Putting semaphore
Worker: Terminated
```

Getting Workers to Work

Sum of First N Integers

The calculation of the sum of the first n integers or $\sum_{i=1}^{n} i$ can be easily calculated in a regular computer program. An ANSI C program would be:

```
#include <stdio.h>
#define N 6

int main{
  int i;
  int sum = 0;

  for(i=N; i>=N; i--)
    sum+=i;

  printf("The sum of the first %d integers is %d\n", N, sum);
  return 0;
}
```

This problem can easily be performed in a parallel program by having the master (tupleSum1Master.c) put each integer into the problem tuple space. The workers (tupleSum1Workers.c) take the integers out of the problem tuple space, tally their respective sub sums and put the sub sums into the result tuple space. The master gets the sub sums from the result tuple space and produces the desires sum. This application is located in the example04 directory.

The following is the tuple space sum of n integers master program:

```
#include <stdio.h>
#include <sys/resource.h>

main(){

  int P;                    // Number of processors
  int i;                    // Counter index
  int status;               // Return status for tuple operations
  int res;                  // Result tuple space identifier
  int tsd;                  // Problem tuple space identifier
  int maxNum = 6;           // MAX of n for sum of 1..n
  int sendNum = 0;          // Number sent to problem ts
  int *sendPtr = &sendNum;  // Pointer to sendNum
  int recdSum = 0;          // Subsum received from result ts
  int *recdPtr = &recdSum;  // Pointer to recdSum
  int calcSum = 0;          // Calculated sum
  int sumTotal = 0;         // Sum total of all subsums
  int tplength;             // Length of ts entry
```

```
char tpname[20];          // Identifier of ts entry
char host[128];           // Host machine name

// Get host machine name
gethostname(host, sizeof(host));

// Open tuple spaces
printf("Master: Opening tuple spaces\n");
// Open problem tuple space
tsd = cnf_open("problem", 0);
// Open result tuple space
res = cnf_open("result", 0);
printf("Master: Tuple spaces open complete\n");

// Get number of processors
P = cnf_getP();
printf("Master: Processors %d\n", P);

// Send integers to problem tuple space
// Set length of entry
tplength = sizeof(int);
printf("Master: tplength = (%d)\n", tplength);
// Set maximum n
sendNum = maxNum;
printf("Master: Putting 1...%d to problem tuple space\n", maxNum);
// Loop until all numbers are sent to workers
while (sendNum > 0) {
  printf("Master: Putting %d\n", sendNum);
  // Set name of entry
  sprintf(tpname,"%d", sendNum);
  // Put entry in problem tuple space
  status = cnf_tsput(tsd, tpname, (char *)sendPtr, tplength);
  // Decrement number to set entry value
  sendNum--;
}
printf("Master: Finished sending 1...%d to tuple space\n", maxNum);

// Insert negative integer tuple as termination signal
printf("Master: Sending terminal signal\n");
// Set length of entry
tplength = sizeof(int);
// Set entry value
sendNum = -1;
// Set entry name
sprintf(tpname, "%d", maxNum+1);
// Put entry in problem tuple space
status = cnf_tsput(tsd, tpname, (char *)sendPtr, tplength);
printf("Master: Finished sending terminal signal\n");

// Receive sub sums from result tuple space
i = 1;
printf("Master: Getting sub sums from result tuple space\n");
while (i <= P){
  // Set name of entry to any
  strcpy(tpname,"*");
  // Get entry from result tuple space
```

```
    tplength = cnf_tsget(res, tpname, (char *)recdPtr, 0);
    printf("Master: Received %d from %s\n", recdSum, tpname);
    // Add result to total
    sumTotal += recdSum;
    // Increment counter
    i++;
  }
  printf("Master: The sum total is: %d\n", sumTotal);

  // Calculate correct answer with math formula
  calcSum = (maxNum*(maxNum+1))/2;
  printf ("Master: The calculated sum is: %d\n", calcSum);

  // Compare results
  if(calcSum == sumTotal)
    printf("Master: The workers gave the correct answer\n");
  else
    printf("Master: The workers gave an incorrect answer\n");

  // Terminate program
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space sum of n integers worker program:

```
#include <stdio.h>
#include <sys/resource.h>

main(){

  // Variable declarations
  int tsd;                  // Problem tuple space identifier
  int res;                  // Result tuple space identifier
  int recdNum = 0;          // Number received to be added
  int *recdPtr = &recdNum;  // Pointer to recdNum
  int sendSum = 0;          // Sum of numbers received
  int *sendPtr = &sendSum;  // Pointer to sendSum
  int status;               // Return status for tuple operations
  int tplength;             // Length of ts entry
  char tpname[20];          // Identifier of ts entry
  char host[128];           // Host machine name

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Worker: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem", 0);
  // Open result tuple space
  res = cnf_open("result", 0);
  printf("Worker: Tuple spaces open complete\n");

  // Loop forever to accumulate sendSum
```

```
  printf("Worker: Beginning to accumulate sum\n");
  while(1){
    // Set name to any
    strcpy(tpname, "*");
    // Get problem from tuple space
    tplength = cnf_tsget(tsd, tpname, (char *)recdPtr, 0);
    printf("Worker: Took item %s\n", tpname);
    // If normal receive
    if(recdNum > 0){
      // Add to sum
      sendSum += recdNum;
      printf("Worker: Present subtotal is %d\n", sendSum);
    }
    // Else terminate worker
    else{
      printf("Worker: Received terminal signal\n");
      // Put terminal message back in problem tuple space
      status = cnf_tsput(tsd, tpname, (char *)recdPtr, tplength);
      // Set length of entry
      tplength = sizeof(int);
      // Set name of entry to host
      sprintf(tpname,"%s", host);
      printf("Worker: Sending sum %d\n", sendSum);
      // Put sum in result tuple space
      status = cnf_tsput(res, tpname, (char *)sendPtr, tplength);
      // Terminate worker
      printf("Worker: Terminated\n");
      cnf_term();
    }
    // Sleep 1 second
    sleep(1);
  }
}
```

To run the sum of first n integers distributed application:

1. Make the executables by typing “make” and pressing the enter key.
2. Run the application by typing “prun tupleSum1” and pressing the enter key.

The screen output for the master terminal with Synergy’s initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc04 ]>prun tupleSum1
Master: Opening tuple spaces
Master: Tuple spaces open complete
Master: Processors 2
Master: tplength = (4)
Master: Putting 1...6 to problem tuple space
Master: Putting 6
Master: Putting 5
Master: Putting 4
Master: Putting 3
Master: Putting 2
```

```
Master: Putting 1
Master: Finished sending 1...6 to tuple space
Master: Sending terminal signal
Master: Finished sending terminal signal
Master: Getting sub sums from result tuple space
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Beginning to accumulate sum
Worker: Took item 5
Worker: Present subtotal is 5
Worker: Took item 3
Worker: Present subtotal is 8
Worker: Took item 1
Worker: Present subtotal is 9
Master: Received 12 from saber
Worker: Took item 7
Worker: Received terminal signal
Worker: Sending sum 9
Worker: Terminated
Master: Received 9 from owin
Master: The sum total is: 21
Master: The calculated sum is: 21
Master: The workers gave the correct answer
Master: Terminated
[c615111@owin ~/fpc04 ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Tuple spaces open complete
Worker: Beginning to accumulate sum
Worker: Took item 6
Worker: Present subtotal is 6
Worker: Took item 4
Worker: Present subtotal is 10
Worker: Took item 2
Worker: Present subtotal is 12
Worker: Took item 7
Worker: Received terminal signal
Worker: Sending sum 12
Worker: Terminated
```

Matrix Multiplication

Matrix multiplication, $A \cdot B = C$, can be performed by a traditional C program using the following function:

```
void multIntMats(int A[N][N], int B[N][N], int C[N][N]){
  int i=0, j=0, k=0;
  for(i=0; i<N; i++){
```

```
        for(j=0; j<N; j++){
              C[i][j] = 0;
              for(k=0; k<N; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
  }
}
```

The program uses nested loops to calculate the value of the elements in the C matrix by performing multiplications across each of the elements in each row of the A matrix with each of the elements in each column of the B matrix. In other words, the value at position C[0][0] is calculated by adding the products across the first row of A with those of the first column B such that:

$$C[0][0] = \sum_{k=0}^{n-1} A[0][k] \cdot B[k][0].$$

And by the function above for an n × n matrix we have:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j],\ 0 \le \text{i} \le \text{n-1},\ 0 \le \text{j} \le \text{n-1}.$$

We can conclude that:

$$C[0][j] = \sum_{k=0}^{n-1} A[0][k] \cdot B[k][j],\ 0 \le \text{j} \le \text{n-1}.$$

Or in terms of the multiplication of a vector by a matrix for the i^{th} rows of A and C:

$$C_i[j] = \sum_{k=0}^{n-1} A_i[k] \cdot B[k][j],\ 0 \le \text{i} \le \text{n-1}, 0 \le \text{j} \le \text{n-1}.$$

A parallel program can perform matrix multiplication by having the master (tupleMat1Master.c) puts the whole matrix B as another single entry and each row of A into the tuple space as a single entry. The worker (tupleMat1Worker.c) reads the whole B matrix and takes individual rows of A out of the problem tuple space. Given an n-length array Ai, which is a single row of A, the whole B matrix and an array Ci to store the result of the procedure, which is a row of C, the worker performs the following (simplified for illustration) procedure on the data:

```
Begin procedure worker multiply
Get B from problem tuple space
While there are arrays in tuple space
  Get an array from problem tuple space and put in Ai
  For i=0..n-1
    Set Ci[i] to 0
    For j=0..n-1
      Ci[i] += Ai[j] * B[j][i]
  Put array Ci in result tuple space
End procedure worker multiply
```

The procedure multiplies an array (or vector) by a matrix. An example of this procedure is:

$$A0 := (1\ 0\ 1\ 0\ 0\ 0) \qquad B = \begin{bmatrix} 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad C0 := A0 \cdot B \qquad C0 = (1\ 0\ 0\ 0\ 0\ 0)$$

$$A := \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix} \quad C := A \cdot B \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad B := A^{-1}$$

The master will know which row to put the Ci results in because the tuple name (the i) will be the row number, which is also the tuple entry name. The multiplication of A and B after the results were taken out of the result tuple space and assembled by the master would be:

Notice that the multiplication produces the identity matrix. The B matrices used in examples are intentionally set to be the inverse of their respective A matrices to demonstrate that the programs actually work. The files for this application are located in the example05 directory. The master program for the matrix multiplication is:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

#define N 6
```

```
main(){

  int i, j;             // Matrix indices
  int tplength;         // Length of ts entry
  int status;           // Return status for tuple operations
  int P;                // Number of processors
  int res;              // Result tuple space identifier
  int tsd;              // Problem tuple space identifier
  int n;                // Counter
  int Ai[N];            // Row from A to send to worker
  int Ci[N];            // Row from C to get from worker
  char host[128];       // Host machine name
  char tpname[20];      // Identifier of ts entry

  // The A matrix to break up into arrays
  // and send to workers
  int A[N][N] = {{1,0,1,0,0,0},
                 {0,1,0,1,0,0},
                 {1,0,1,0,1,0},
                 {0,1,0,1,0,1},
                 {0,0,1,0,1,0},
                 {0,0,0,1,0,1}};
  // The B matrix to send to workers
  int B[N][N] = {{0,0,1,0,-1,0},
                 {0,0,0,1,0,-1},
                 {1,0,-1,0,1,0},
                 {0,1,0,-1,0,1},
                 {-1,0,1,0,0,0},
                 {0,-1,0,1,0,0}};
  // The C matrix built from arrays
  // received from workers
  int C[N][N];

  printf("Master: started\n");

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Master: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  // Open result tuple space
  res = cnf_open("result",0);
  printf("Master: Tuple spaces open complete\n");

  // Get number of processors
  P = cnf_getP(); // Get number of processors
  printf("Master: Processors %d\n", P);

  // Print matrix A and B
  printf("Master: Matrix A\n");
  for(i=0; i<N; i++){
        for(j=0; j<N; j++){
              printf(" %d", A[i][j]);
        }
```

```
      printf("\n");
}
printf("Master: Matrix B\n");
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    printf(" %d", B[i][j]);
  }
  printf("\n");
}

printf("Master: Starting C = A . B\n");

// Put B matrix in ts
// Set length of send entry
tplength = N*N*sizeof(int);
// Set name of entry to B
sprintf(tpname,"B",0);
printf("Master: Putting Length %d Name %s\n", tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, B, tplength);

// Put A matrix in ts
// Set length of send entry
tplength = N*sizeof(int);
printf("tplength =  %d\n", tplength);

// Ready to build Ai row
for (i = 0; i < N; i++){
  // Set the rows name to row index
  sprintf(tpname,"A%d",i);
  printf("Master: Putting item %s ", tpname);
  // Put a row from A matrix in ituple_A array
  for (j = 0; j < N; j++){
    Ai[j] = A[i][j];
    printf("%d ", Ai[j]);
  }
  printf("\n");
  // Put entry in tuple space
  status = cnf_tsput(tsd, tpname, Ai, tplength);
}

// Build C matrix from workers' results
for(n=0; n<N; n++){
  // Set name to any
  strcpy(tpname,"*");
  // Read result from result tuple space
  tplength = cnf_tsget(res, tpname, Ci, 0);
  printf("Master: Received %s\n", tpname);
  // Set received row to tpname
  i = atoi(tpname);
  // Add this array to C
  for (j=0; j<N; j++) {
    C[i][j] = Ci[j];
  }
}
```

```
  // Print the C matrix from workers
  printf("Master: Matrix C\n");
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      printf(" %d", C[i][j]);
    }
    printf("\n");
  }

  // Insert -2 tuple as termination signal
  // Set length of send entry
  printf("Master: Putting terminal signal\n");
  tplength = sizeof(int);
  Ai[0]  = -2;
  sprintf(tpname, "A%d",N);
  status = cnf_tsput(tsd, tpname, Ai, tplength);

  // Terminate master
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space matrix multiplication worker program:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

#define N 6

main(){

  int tsd;              // Problem tuple space identifier
  int res;              // Result tuple space identifier
  int i, j;             // Matrix indices
  int n;                // Counter
  int status;           // Return status for tuple operations
  int tplength;         // Length of ts entry
  int Ai[N];            // Row from A to get from master
  int Ci[N];            // Column from C to send to master
  int B[N][N];          // B matrix received from master
  char host[128];       // Host machine name
  char tpname[20];      // Identifier of ts entry

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Worker: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  // Open result tuple space
  res = cnf_open("result",0);
  printf("Worker: Tuple spaces open complete\n");
```

```
// Set name to B
strcpy(tpname,"B");
// Read B matrix from problem tuple space
status = cnf_tsread(tsd, tpname, B, 0);
tplength = (N*N)*sizeof(double);

printf("Worker: Matrix B\n");
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    printf(" %d", B[i][j]);
  }
  printf("\n");
}

// Loop forever to get work
while(1){
  // Set name to any
  strcpy(tpname,"A*");
  // Read problem from problem tuple space
  tplength = cnf_tsget(tsd, tpname, Ai, 0);
  printf("Worker: Taking item %s", tpname);

  // Normal receive
  if(tplength > 0){
    printf("\n");
    // Check for the application termination signal
    if (Ai[0] < -1){
      // Replace the terminal signal in problem ts
      status = cnf_tsput(tsd, tpname, Ai, tplength);
      printf("Worker: Terminated s\n");
      cnf_term();
    }

    for(i=0; i<N; i++)
      printf(" %d", Ai[i]);
    printf("\n");

    // Perform multiplication on array and matrix
    for(i=0; i<N; i++){
      Ci[i] = 0;
      for(j=0; j<N; j++)
        Ci[i] += Ai[j]*B[j][i];
    }

    // Get row number
    i = atoi(&tpname[1]);

    // Print the result array
    printf("Worker : Array C%s", tpname);
    for(n=0; n<N; n++)
      printf(" %d", Ci[n]);
    printf("\n");

    // Set name to row number
    sprintf(tpname,"%d",i);
    // Put the result in the result tuple space
```

```
      status = cnf_tsput(res, tpname, Ci, tplength);
      sleep(1);
    }

    // Else a zero length tuple was received
    else{
      printf("Worker: Error-received zero length tuple");
      printf("Worker: Terminated\n");
      cnf_term();
    }
  }
}
```

To run the matrix multiplication distributed application:

1. Make the executables by typing “make” and pressing the enter key.
2. Run the application by typing “prun tupleMat1” and pressing the enter key.

The screen output for the master terminal with Synergy’s initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc05 ]>prun tupleMat1
Master: Tuple spaces open complete
Master: Processors 2
Master: Matrix A
 1 0 1 0 0 0
 0 1 0 1 0 0
 1 0 1 0 1 0
 0 1 0 1 0 1
 0 0 1 0 1 0
 0 0 0 1 0 1
Master: Matrix B
 0 0 1 0 -1 0
 0 0 0 1 0 -1
 1 0 -1 0 1 0
 0 1 0 -1 0 1
 -1 0 1 0 0 0
 0 -1 0 1 0 0
Master: Starting C = A . B
Master: Putting Length 144 Name B
Master: tplength =  24
Master: Putting item A0 1 0 1 0 0 0
Master: Putting item A1 0 1 0 1 0 0
Master: Putting item A2 1 0 1 0 1 0
Master: Putting item A3 0 1 0 1 0 1
Master: Putting item A4 0 0 1 0 1 0
Master: Putting item A5 0 0 0 1 0 1
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Matrix B
 0 0 1 0 -1 0
 0 0 0 1 0 -1
Master: Received 0
```

```
 1 0 -1 0 1 0
 0 1 0 -1 0 1
 -1 0 1 0 0 0
 0 -1 0 1 0 0
Worker: Taking item A1
 0 1 0 1 0 0
Worker : Array CA1 0 1 0 0 0 0
Master: Received 1
Worker: Taking item A2
 1 0 1 0 1 0
Worker : Array CA2 0 0 1 0 0 0
Master: Received 2
Master: Received 3
Worker: Taking item A4
 0 0 1 0 1 0
Worker : Array CA4 0 0 0 0 1 0
Master: Received 4
Master: Received 5
Master: Matrix C
 1 0 0 0 0 0
 0 1 0 0 0 0
 0 0 1 0 0 0
 0 0 0 1 0 0
 0 0 0 0 1 0
 0 0 0 0 0 1
Master: Putting terminal signal
Master: Terminated
Worker: Taking item A6
Worker: Terminated
[c615111@owin ~/fpc05 ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Matrix B
 0 0 1 0 -1 0
 0 0 0 1 0 -1
 1 0 -1 0 1 0
 0 1 0 -1 0 1
 -1 0 1 0 0 0
 0 -1 0 1 0 0
Worker: Taking item A0
 1 0 1 0 0 0
Worker : Array CA0 1 0 0 0 0 0
Worker: Taking item A3
 0 1 0 1 0 1
Worker : Array CA3 0 0 0 1 0 0
Worker: Taking item A5
 0 0 0 1 0 1
Worker : Array CA5 0 0 0 0 0 1
Worker: Taking item A6
Worker: Terminated
```

Work Distribution by Chunking

Finding the Sum of the First n Integers with Chunking

The following is the tuple space "sum of n integers" master program implemented by sending work in chunks:

```
#include <stdio.h>
#include <sys/resource.h>

#define N 32

main(){

  int P;                    // Number of processors
  int chunk_size;           // Chunk size
  int remainder;            // Remainder of numbers to be sent
  int i;                    // Counter index
  int job;                  // Job number
  int status;               // Return status for tuple operations
  int res;                  // Result tuple space identifier
  int tsd;                  // Problem tuple space identifier
  int *sendArr = 0;         // Number sent to problem ts
  int sendNum;              // Number sent to worker in sendArr
  int recdSum = 0;          // Subsum recieved from result ts
  int *recdPtr = &recdSum;  // Pointer to recdSum
  int calcSum = 0;          // Calculated sum
  int sumTotal = 0;         // Sum total of all subsums
  int tplength;             // Length of ts entry
  char tpname[20];          // Identifier of ts entry
  char host[128];           // Host machine name

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Master: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem", 0);
  // Open result tuple space
  res = cnf_open("result", 0);
  printf("Master: Tuple spaces open complete\n");

  // Get number of processors
  P = cnf_getP();
  printf("Master: Processors %d\n", P);
  // Get chunk size
  chunk_size = cnf_getf();
  printf("Master: Chunk size %d\n", chunk_size);

  // Put chunk size in ts
  // Set length of entry
```

```
tplength = sizeof(int);
// Set name of entry
strcpy(tpname, "chunk_size");
// Put entry in ts
status = cnf_tsput(tsd, tpname, &chunk_size, tplength);
printf("Master: Sent chunk size\n");

// Send integers to problem tuple space
// Set length of entry to chunk_size + 1 integers
tplength = (chunk_size+1) * sizeof(int);
printf("Master: tplength = %d\n", tplength);

// Prepare and send integer arrays into tuple space
printf("Master: Putting 1...%d to problem tuple space\n", N);
if((sendArr = (int *) malloc(tplength)) == NULL)
  exit(1);
// Loop until all numbers are sent to workers
remainder = N;
job = 0;
sendNum = 1;
while (remainder > 0) {
  if (remainder < chunk_size)
    chunk_size = remainder;
  remainder = remainder - chunk_size;
  job++;
  // Set name of entry to job number
  sprintf(tpname,"A%d", job);
  // Put chunk_size in index zero
  sendArr[0] = chunk_size;
  printf("Master: Putting %s Size %d\n ", tpname, sendArr[0]);
  // Put chunk_size integers in array
  for(i=1; i<=chunk_size; i++, sendNum){
    sendArr[i] = sendNum++;
    printf(" %d", sendArr[i]);
  }
  printf("\n");
  // Put entry in problem tuple space
  status = cnf_tsput(tsd, tpname, sendArr, tplength);
  // Decrement number to set entry value
}
printf("Master: Finished sending 1...%d to tuple space\n", N);

// Receive sub sums from result tuple space
// Set index to 1
i = 1;
printf("Master: Getting sub sums from result tuple space\n");
while (job-- > 0){
  // Set name of entry to any
  strcpy(tpname,"*");
  // Get entry from result tuple space
  tplength = cnf_tsget(res, tpname, (char *)recdPtr, 0);
  printf("Master: Recieved %d from %s\n", recdSum, tpname);
  // Add result to total
  sumTotal += recdSum;
  // Increment counter
}
```

```
  printf("Master: The sum total is: %d\n", sumTotal);

  // Calculate correct answer with math formula
  calcSum = (N*(N+1))/2;
  printf ("Master: The formula calculated sum is: %d\n", calcSum);

  // Compare results
  if(calcSum == sumTotal)
    printf("Master: The workers gave the correct answer\n");
  else
    printf("Master: The workers gave an incorrect answer\n");

  // Insert negative integer tuple as termination signal
  printf("Master: Sending terminal signal\n");
  // Set length of entry
  tplength = (1) * sizeof(int);
  // Set entry value
  sendArr[0] = -1;
  // Set entry name
  sprintf(tpname, "A%d", N+1);
  // Send entry to tuple space
  status = cnf_tsput(tsd, tpname, sendArr, tplength);
  printf("Master: Finished sending terminal signal\n");

  // Terminate program
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space “sum of n integers” worker program implemented by receiving work in chunks:

```
#include <stdio.h>
#include <sys/resource.h>

main(){

  // Variable declarations
  int tsd;                  // Problem tuple space identifier
  int res;                  // Result tuple space identifier
  int *recdPtr;             // Pointer to recd array
  int sendSum = 0;          // Sum of numbers received
  int *sendPtr = &sendSum;  // Pointer to sendSum
  int status;               // Return status for tuple operations
  int tplength;             // Length of ts entry
  int chunk_size;           // Size of recdPtr
  int i;                    // Index counter
  char tpname[20];          // Identifier of ts entry
  char host[128];           // Host machine name

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
```

```
printf("Worker: Opening tuple spaces\n");
// Open problem tuple space
tsd = cnf_open("problem", 0);
// Open result tuple space
res = cnf_open("result", 0);
printf("Worker: Tuple spaces open complete\n");

// Get the chunk size from ts
// Set name of entry
strcpy(tpname, "chunk_size");
// Read chunk size
status = cnf_tsread(tsd, tpname, &chunk_size, 0);
printf("Worker: Chunk size %d\n", chunk_size);

// Set length of tuple space entry
tplength = (chunk_size+1) * sizeof(int);
// Allocate memory for entry
if((recdPtr = (int *)malloc(tplength)) == NULL)
  exit(-1);
printf("Worker: array size %d\n", tplength);

// Loop forever to accumulate sendSum
printf("Worker: Begining to accumulate sum\n");
while(1){
  sendSum = 0;
  // Set name to any
  strcpy(tpname, "A*");
  // Get problem from tuple space
  tplength = cnf_tsget(tsd, tpname, recdPtr, 0);
  // Get chunk_size from index zero
  chunk_size = (int) recdPtr[0];
  printf("Worker: Took item %s length %d\n ", tpname, chunk_size);
  // If normal receive
  if(chunk_size > 0){
    // Get number of array elements
    // Add to sendSum
    for(i=1; i<=chunk_size; i++){
      sendSum += recdPtr[i];
      printf(" %d", recdPtr[i]);
    }
    // Set length of entry
    tplength = sizeof(int);
    // Set name of entry to host
    strcpy(tpname, host);
    printf("\nWorker: Sending sum %d\n", sendSum);
    // Put sum in result tuple space
    status = cnf_tsput(res, tpname, sendPtr, tplength);
  }
  // Else terminate worker
  else{
    printf("Worker: Recieved terminal signal\n");
    // Put terminal message back in problem tuple space
    status = cnf_tsput(tsd, tpname, recdPtr, tplength);
    // Terminate worker
    printf("Worker: Terminated\n");
    cnf_term();
```

```
    }
    // Sleep 1 second
    sleep(1);
  }
}
```

To run the sum of first n integers distributed application with chunking:

1. Make the executables by typing “make” and pressing the enter key.
2. Run the application by typing “prun tupleSum2” and pressing the enter key.

The screen output for the master terminal with Synergy’s initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc06 ]>prun tupleSum2
Master: Opening tuple spaces
Master: Tuple spaces open complete
Master: Processors 2
Master: Chunk size 4
Master: Sent chunk size
Master: tplength = 20
Master: Putting 1...32 to problem tuple space
Master: Putting A1 Size 4
  1 2 3 4
Master: Putting A2 Size 4
  5 6 7 8
Master: Putting A3 Size 4
  9 10 11 12
Master: Putting A4 Size 4
  13 14 15 16
Master: Putting A5 Size 4
  17 18 19 20
Master: Putting A6 Size 4
  21 22 23 24
Master: Putting A7 Size 4
  25 26 27 28
Master: Putting A8 Size 4
  29 30 31 32
Master: Finished sending 1...32 to tuple space
Master: Getting sub sums from result tuple space
Master: Recieved 10 from saber
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Chunk size 4
Worker: array size 20
Worker: Begining to accumulate sum
Worker: Took item A2 length 4
  5 6 7 8
Worker: Sending sum 26
Master: Recieved 26 from owin
Master: Recieved 42 from saber
Worker: Took item A4 length 4
  13 14 15 16
```

```
Worker: Sending sum 58
Master: Recieved 58 from owin
Master: Recieved 74 from saber
Worker: Took item A6 length 4
  21 22 23 24
Worker: Sending sum 90
Master: Recieved 90 from owin
Master: Recieved 106 from saber
Worker: Took item A8 length 4
  29 30 31 32
Worker: Sending sum 122
Master: Recieved 122 from owin
Master: The sum total is: 528
Master: The formula calculated sum is: 528
Master: The workers gave the correct answer
Master: Sending terminal signal
Master: Finished sending terminal signal
Master: Terminated
Worker: Took item A33 length -1
Worker: Recieved terminal signal
Worker: Terminated
[c615111@owin ~/fpc06 ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Chunk size 4
Worker: array size 20
Worker: Begining to accumulate sum
Worker: Took item A1 length 4
  1 2 3 4
Worker: Sending sum 10
Worker: Took item A3 length 4
  9 10 11 12
Worker: Sending sum 42
Worker: Took item A5 length 4
  17 18 19 20
Worker: Sending sum 74
Worker: Took item A7 length 4
  25 26 27 28
Worker: Sending sum 106
Worker: Took item A33 length -1
 Worker: Recieved terminal signal
Worker: Terminated
```

Matrix Multiplication with Chunking

A =

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	0
2	1	1	1	1	1	1	1	1	0	0
3	1	1	1	1	1	1	1	0	0	0
4	1	1	1	1	1	1	0	0	0	0
5	1	1	1	1	1	0	0	0	0	0
6	1	1	1	1	0	0	0	0	0	0
7	1	1	1	0	0	0	0	0	0	0
8	1	1	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0

B =

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1	-1
2	0	0	0	0	0	0	0	1	-1	0
3	0	0	0	0	0	0	1	-1	0	0
4	0	0	0	0	0	1	-1	0	0	0
5	0	0	0	0	1	-1	0	0	0	0
6	0	0	0	1	-1	0	0	0	0	0
7	0	0	1	-1	0	0	0	0	0	0
8	0	1	-1	0	0	0	0	0	0	0
9	1	-1	0	0	0	0	0	0	0	0

$C := A \cdot B$

C =

	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

$B := A^{-1}$

The following is the tuple space “matrix multiplication” master program implemented by sending work in chunks:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "matrix.h"

// The A matrix to break up into arrays
// and send to workers
double A[N][N];
// The B matrix
double B[N][N];
// The resulting C matrix
double C[N][N];
```

```
main(){

  int processors;     // Number of processors
  int chunk_size;     // Chunk size
  int remaining;      // Remaining arrays of work
  int i, j;           // Matrix indices
  int matrix_row;     // Index of matrix row
  int array_pos;      // Array position in rows array
  int status;         // Return status for tuple operations
  int res;            // Result tuple space identifier
  int tsd;            // Problem tuple space identifier
  double *rows;       // Rows from A to send to worker
  double worker_time; // Sum of times returned by workers
  double total_time;  // Total application run time
  int tplength;       // Length of ts entry
  char tpname[20];    // Identifier of ts entry
  char host[128];     // Host machine name


  // Get host machine name
  gethostname(host, sizeof(host));

  // Get time stamp
  total_time = wall_clock();

  // Open tuple spaces
  printf("Master: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  // Open result tuple space
  res = cnf_open("result",0);
  printf("Master: Tuple spaces open complete\n");

  // Get number of processors
  processors = cnf_getP();
  printf("Master: Processors %d\n", processors);
  // Get chunk size
  chunk_size = cnf_getf();
  printf("Master: Chunk size %d\n", chunk_size);

  printf("Master: Starting C = A . B\n");
  printf("  on %d x %d matrices\n", N, N);

  // Create and print matrix B
  makeDblInv(B);
  if(N <= 36)
    printDblMat(B, 'B');

  // Put B matrix in ts
  // Set size of B matrix
  tplength = N*N*sizeof(double);
  // Set name of entry to B
  sprintf(tpname,"B",0);
  printf("Master: Putting B Length(%d) Name(%s)\n",
         tplength, tpname);
```

```
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, B, tplength);

// Create and print matrix A
makeDblMat(A);
if(N <= 36)
  printDblMat(A, 'A');

// Put chunk_size of A in ts
// Set size of int
tplength = sizeof(int);
// Set name of entry to chunk_size
sprintf(tpname,"chunk_size",0);
printf("Master: Putting chunk_size Length(%d) Name(%s)\n",
        tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, &chunk_size, tplength);

// Put chunks of A in ts
// Get Ai tuple size
tplength = (2+chunk_size*N) * sizeof(double);
printf("Master: Ai tplength = (%d)\n", tplength);

// Prepare integer array rows for tuple space exchange
if((rows = (double *) malloc(tplength)) == NULL)
  exit(1);
printf("Master: Putting A in problem tuple space\n");

// Build Ai array rows to send to ts
// Set remaining to total number of rows
remaining = N;
// Set start matrix row to zero
matrix_row = 0;
// Loop until all numbers are sent to workers
while (remaining > 0) {
  // If remaining rows is less than chunk size
  // set number of rows sent to remaining rows
  if (remaining < chunk_size)
    chunk_size = remaining;
  // Subtract rows being sent from remaining rows
  remaining = remaining - chunk_size;
  printf("Master: chunk_size(%d) remaining(%d) \n",
          chunk_size, remaining);
  // Put chunk_size in first index
  rows[0] = chunk_size;
  // Set rows array position to 2
  // Second position (1) is reserved for
  // time returned by worker
  array_pos = 2;
  // Put rows of A matrix in rows array
  for (i=0; i<chunk_size; i++){
    for (j=0; j<N; j++){
      rows[array_pos] = A[matrix_row+i][j];
      if(N <= 36)
        printf(" %g", rows[array_pos]);
      array_pos++;
```

```
    }
    if(N <= 36)
      printf("\n");
  }
  // Set entry name to beginning Ai-row
  sprintf(tpname,"A%d",matrix_row);
  status = cnf_tsput(tsd, tpname, rows, tplength);
  matrix_row += chunk_size;
}

// Get the result Ci from ts and assemble
// Set received rows to zero
remaining = N;
// Initialize worker time
worker_time = 0;
// Loop until all rows are recieved
while(remaining > 0){
  // Set entry name
  strcpy(tpname,"*");
  // Get entry from result tuple space
  tplength = cnf_tsget(res, tpname, rows, 0);
  // Get number rows in this chunk from last index
  chunk_size = rows[0];
  // Get time returned by worker
  worker_time += rows[1];
  // Convert beginning row of entry to an integer
  matrix_row = atoi(tpname);
  printf("Master: Recieved %s Size %d\n", tpname, chunk_size);
  // Set the position in the array to 2
  array_pos = 2;

  // Assemble the result matrix C
  // Loop through recieved rows
  printf("Master: Recieved\n");
  for (i= 0; i<chunk_size; i++){
    // Increment rows recieved by decrementing remaining
    remaining--;
    // Loop through row and array elements
    for (j=0; j<N; j++){
      C[matrix_row][j] = rows[array_pos];
      if(N <= 36)
        printf(" %g", C[matrix_row][j]);
      // Increment array position
      array_pos++;
    }
    if(N <= 36)
      printf("\n");
    // Increment row position
    matrix_row++;
  }
}

// Resolve total time
total_time = wall_clock() - total_time;
printf("Master: The multiplication took %g seconds total time\n",
       (total_time/1000000.0));
```

```
  // Resolve worker time
  printf("Master: The workers used %g seconds of processor time\n",
         (worker_time/1000000.0));

  // Check and print the C matrix
  if(N <= 36)
    printDblMat(C, 'C');
  checkDblIdenMat(C, 'C');

  // Insert termination signal
  // Set length of entry
  tplength = sizeof(double);
  // Set entry value
  i  = -1;
  // Set entry name
  strcpy(tpname, "A-term");
  // Send entry to tuple space
  status = cnf_tsput(tsd, tpname, &i, tplength);

  // Free memory for rows array
  free(rows);

  // Terminate program
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space "matrix multiplication" worker program implemented by sending work in chunks:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "matrix.h"

double B[N][N];   // B matrix
double Ai[N];     // N length row of A

main(){

  int chunk_size;     // Chunk size
  int remaining;      // Remaining arrays of work
  int i, j;           // Matrix indices
  int n;              // Counter for rows in chunk
  int matrix_row;     // Index of matrix row
  int array_get;      // Get array position in rows array
  int array_put;      // Put array position in rows array
  int status;         // Return status for tuple operations
  int res;            // Result tuple space identifier
  int tsd;            // Problem tuple space identifier
  double *rows;       // Rows from A to send to worker
  double worker_time; // Time to return to master
```

```
int tplength;       // Length of ts entry
char tpname[20];    // Identifier of ts entry
char host[128];     // Host machine name

// Get host machine name
gethostname(host, sizeof(host));

// Open tuple spaces
printf("Worker: Opening tuple spaces\n");
// Open problem tuple space
tsd = cnf_open("problem", 0);
// Open result tuple space
res = cnf_open("result", 0);
printf("Worker: Tuple spaces open complete\n");

// Set tpname to B
strcpy(tpname,"B");
// Read matrix B from tuple space
status = cnf_tsread(tsd, tpname, B, 0);

// Print matrix B
if(N <= 36)
  printDblMat(B, 'B');

// Get chunk_size from master
// Set tpname to chunk_size
strcpy(tpname,"chunk_size");
// Read chunk_size from tuple space
status = cnf_tsread(tsd, tpname, &chunk_size, 0);

// Prepare integer array for tuple space exchanges
tplength = (1+chunk_size*N)*sizeof(double);
if ((rows = (double*)malloc(tplength)) == NULL)
  exit(-1);

// Loop until terminal signal is recieved
while(1){
  // Set entry name
  strcpy(tpname,"A*");
  // Set length of entry
  tplength = cnf_tsget(tsd, tpname, rows, 0);

  // Normal recieve
  if(tplength > 0){
    // Check termination signal
    if (!strcmp(tpname, "A-term")){
      printf("Worker: Recieved the terminal signal\n");
      // Replace the terminal signal in problem ts
      status = cnf_tsput(tsd, tpname, rows, tplength);
      // Free memory for rows
      free(rows);
      // Terminate worker
      printf("Worker: Terminated\n");
      cnf_term();
    }
    // Get number rows in this chunk from last index
```

```
chunk_size = rows[0];
// Convert beginning row of entry to an integer
matrix_row = atoi(&tpname[1]);
printf("Worker: chunk_size %d matrix_row %d\n",
       chunk_size, matrix_row);

// Set rows array put position to 2
array_put = 2;
// Set rows array get position to 2
array_get = 2;

// Get beginning worker time
worker_time = wall_clock();

// For each row in chunk_size
for(n=0; n<chunk_size; n++){
  // Copy a row from rows to Ai
  // and print to screen
  if(N <= 36)
    printf("Worker: Recieved\n");
  for(i=0; i<N; i++){
    Ai[i] = rows[array_get];
    rows[array_get++] = 0;
    if(N <= 36)
      printf(" %g", Ai[i]);
  }
  if(N <= 36)
    printf("\n");
  // Multiply rows in place with B
  // For each column of B
  if(N <= 100)
    printf("Worker: Calculated array C%s+%d\n", tpname, n);
  for (i=0; i<N; i++){
    // For each element in Ai and each
    // element in this column of B multiply
    // producing an element in rows
    for (j=0; j<N; j++){
      rows[array_put] += Ai[j] * B[j][i];
    }
    if(N <= 36)
      printf(" %g", rows[array_put]);
    // Increment to next position in rows
    array_put++;
  }
  if(N <= 36)
    printf("\n");
}

// Put worker time in rows array
rows[1] = wall_clock() - worker_time;
// Set length of entry
tplength = (2+chunk_size*N)*sizeof(double);
// Set tpname to first row number in rows
sprintf(tpname,"%d", matrix_row);
printf("Worker: Putting %s\n",tpname);
// Put the result in the result tuple space
```

```
      status = cnf_tsput(res, tpname, rows, tplength);
      if(N <= 36)
        sleep(1);
    }

    else{
      printf("Worker: Recieved a zero length entry\n");
      // Free memory for rows
      free(rows);
      // Terminate worker
      printf("Worker Terminated\n");
      cnf_term();
    }
  }
}
```

To run the matrix multiplication distributed application with chunk size of 4 and N = 10 (a 10 x 10 matrix):

1. Set the factor value in the csl file to 4 (as shown below)
2. Make the executables by typing "make SIZE=10" and pressing the enter key.
3. Run the application by typing "prun tupleSum2" and pressing the enter key.

```
configuration: tupleMat2;

m: master = tupleMat2Master
        (factor = 4
        threshold = 1
        debug = 0
        )
   -> f: problem
        (type = TS)
   -> m: worker = tupleMat2Worker
        (type = slave)
   -> f: result
        (type = TS)
   -> m: master;
```

The screen output for the master terminal with Synergy's initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc07new ]>prun tupleMat2
Master: Opening tuple spaces
Master: Tuple spaces open complete
Master: Processors 2
Master: Chunk size 4
Master: Starting C = A . B
  on 10 x 10 matrices
The B double matrix
 0 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 0 1 -1
```

```
 0 0 0 0 0 0 0 1 -1 0
 0 0 0 0 0 0 1 -1 0 0
 0 0 0 0 0 1 -1 0 0 0
 0 0 0 0 1 -1 0 0 0 0
 0 0 0 1 -1 0 0 0 0 0
 0 0 1 -1 0 0 0 0 0 0
 0 1 -1 0 0 0 0 0 0 0
 1 -1 0 0 0 0 0 0 0 0
Master: Putting B Length(800) Name(B)
The A double matrix
 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 0
 1 1 1 1 1 1 1 1 0 0
 1 1 1 1 1 1 1 0 0 0
 1 1 1 1 1 1 0 0 0 0
 1 1 1 1 1 0 0 0 0 0
 1 1 1 1 0 0 0 0 0 0
 1 1 1 0 0 0 0 0 0 0
 1 1 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 0
Master: Putting chunk_size Length(4) Name(chunk_size)
Master: Ai tplength = (336)
Master: Putting A in problem tuple space
Master: chunk_size(4) remaining(6)
 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 0
 1 1 1 1 1 1 1 1 0 0
 1 1 1 1 1 1 1 0 0 0
Master: chunk_size(4) remaining(2)
 1 1 1 1 1 1 0 0 0 0
 1 1 1 1 1 0 0 0 0 0
 1 1 1 1 0 0 0 0 0 0
 1 1 1 0 0 0 0 0 0 0
Master: chunk_size(2) remaining(0)
 1 1 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 0
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
The B double matrix
 0 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 0 1 -1
 0 0 0 0 0 0 0 1 -1 0
 0 0 0 0 0 0 1 -1 0 0
 0 0 0 0 0 1 -1 0 0 0
 0 0 0 0 1 -1 0 0 0 0
 0 0 0 1 -1 0 0 0 0 0
 0 0 1 -1 0 0 0 0 0 0
 0 1 -1 0 0 0 0 0 0 0
 1 -1 0 0 0 0 0 0 0 0
Worker: chunk_size 4 matrix_row 4
Worker: Recieved
 1 1 1 1 1 1 0 0 0 0
Worker: Calculated array CA4+0
 0 0 0 0 1 0 0 0 0 0
Worker: Recieved
 1 1 1 1 1 0 0 0 0 0
```

```
Worker: Calculated array CA4+1
 0 0 0 0 0 1 0 0 0 0
Worker: Recieved
 1 1 1 1 0 0 0 0 0 0
Worker: Calculated array CA4+2
 0 0 0 0 0 0 1 0 0 0
Worker: Recieved
 1 1 1 0 0 0 0 0 0 0
Worker: Calculated array CA4+3
 0 0 0 0 0 0 0 1 0 0
Worker: Putting 4
Master: Recieved 4 Size 4
Master: Recieved
 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0
Master: Recieved 0 Size 4
Master: Recieved
 1 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0
Worker: chunk_size 2 matrix_row 8
Worker: Recieved
 1 1 0 0 0 0 0 0 0 0
Worker: Calculated array CA8+0
 0 0 0 0 0 0 0 0 1 0
Worker: Recieved
 1 0 0 0 0 0 0 0 0 0
Worker: Calculated array CA8+1
 0 0 0 0 0 0 0 0 0 1
Worker: Putting 8
Master: Recieved 8 Size 2
Master: Recieved
 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0 1
Master: The multiplication took 1.11439 seconds total time
Master: The workers used 0.024033 seconds of processor time
The C double matrix
 1 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0 1
Master: C is Identity Matrix
Master: Terminated
Worker: Recieved the terminal signal
Worker: Terminated
== (tupleMat2) completed. Elapsed [2] Seconds.
[c615111@owin ~/fpc07new ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
The B double matrix
 0 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 0 1 -1
 0 0 0 0 0 0 0 1 -1 0
 0 0 0 0 0 0 1 -1 0 0
 0 0 0 0 0 1 -1 0 0 0
 0 0 0 0 1 -1 0 0 0 0
 0 0 0 1 -1 0 0 0 0 0
 0 0 1 -1 0 0 0 0 0 0
 0 1 -1 0 0 0 0 0 0 0
 1 -1 0 0 0 0 0 0 0 0
Worker: chunk_size 4 matrix_row 0
Worker: Recieved
 1 1 1 1 1 1 1 1 1 1
Worker: Calculated array CA0+0
 1 0 0 0 0 0 0 0 0 0
Worker: Recieved
 1 1 1 1 1 1 1 1 1 0
Worker: Calculated array CA0+1
 0 1 0 0 0 0 0 0 0 0
Worker: Recieved
 1 1 1 1 1 1 1 1 0 0
Worker: Calculated array CA0+2
 0 0 1 0 0 0 0 0 0 0
Worker: Recieved
 1 1 1 1 1 1 1 0 0 0
Worker: Calculated array CA0+3
 0 0 0 1 0 0 0 0 0 0
Worker: Putting 0
Worker: Recieved the terminal signal
Worker: Terminated
```

To run the matrix multiplication distributed application with chunk size of 200 and N = 500 (a 500 x 500 matrix):

1. Set the factor value in the csl file to 200 (as shown below)
2. Make the executables by typing "make SIZE=500" and pressing the enter key.
3. Run the application by typing "prun tupleMat2" and pressing the enter key.

```
configuration: tupleMat2;

m: master = tupleMat2Master
        (factor = 200
        threshold = 1
        debug = 0
```

```
      )
   -> f: problem
      (type = TS)
   -> m: worker = tupleMat2Worker
      (type = slave)
   -> f: result
      (type = TS)
   -> m: master;
```

The screen output for the master terminal with Synergy's initialization and termination output removed should resemble:

```
[c615111@owin ~/fpc07new ]>prun tupleMat2
Master: Opening tuple spaces
CID starting program. path (bin/tupleMat2Worker)
Master: Tuple spaces open complete
Master: Processors 2
Master: Chunk size 200
Master: Starting C = A . B
  on 500 x 500 matrices
Master: Putting B Length(2000000) Name(B)
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Master: Putting chunk_size Length(4) Name(chunk_size)
Master: Ai tplength = (800016)
Master: Putting A in problem tuple space
Master: chunk_size(200) remaining(300)
Master: chunk_size(200) remaining(100)
Worker: chunk_size 200 matrix_row 200
Master: chunk_size(100) remaining(0)
Worker: Putting 200
Master: Recieved 0 Size 200
Master: Recieved
Master: Recieved 200 Size 200
Master: Recieved
Master: Recieved 400 Size 100
Master: Recieved
Master: The multiplication took 9.66808 seconds total time
Master: The workers used 15.0322 seconds of processor time
Master: C is Identity Matrix
Worker: Recieved the terminal signal
Master: Terminated
Worker: Terminated
== (tupleMat2) completed. Elapsed [10] Seconds.
 [c615111@owin ~/fpc07new ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: chunk_size 200 matrix_row 0
Worker: Putting 0
```

```
Worker: chunk_size 100 matrix_row 400
Worker: Putting 400
Worker: Recieved the terminal signal
Worker: Terminated
```

Optimized Programs

Optimized Matrix Multiplication with Chunking

The following is the tuple space "optimized matrix multiplication" master program implemented by sending work in chunks:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

// The A matrix to break up into arrays
// and send to workers
double A[N][N];
double B[N][N];
double C[N][N];

#include "matrix.h"


// Main function
main(){

  int processors;     // Number of processors
  int chunk_size;     // Chunk size
  int remaining;      // Remaining arrays of work
  int i, j;           // Matrix indices
  int matrix_row;     // Index of matrix row
  int array_pos;      // Array position in rows array
  int status;         // Return status for tuple operations
  int res;            // Result tuple space identifier
  int tsd;            // Problem tuple space identifier
  double *rows;       // Rows from A to send to worker
  double worker_time; // Sum of times returned by workers
  double total_time;  // Total application run time
  int tplength;       // Length of ts entry
  char tpname[20];    // Identifier of ts entry
  char host[128];     // Host machine name


  // Get host machine name
  gethostname(host, sizeof(host));

  // Get time stamp
  total_time = wall_clock();

  // Open tuple spaces
  printf("Master: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem",0);
  // Open result tuple space
  res = cnf_open("result",0);
  printf("Master: Tuple spaces open complete\n");
```

```
// Get number of processors
processors = cnf_getP();
printf("Master: Processors %d\n", processors);
// Get chunk size
chunk_size = cnf_getf();
printf("Master: Chunk size %d\n",
        chunk_size);

printf("Master: Starting C = A . B\n");
printf("  on %d x %d matrices\n", N, N);

// Create and print matrix B
makeDblInv(B);
if(N <= 36)
  printDblMat(B, 'B');

// Put B matrix in ts
// Set size of B matrix
tplength = N*N*sizeof(double);
// Set name of entry to B
sprintf(tpname,"B",0);
printf("Master: Putting B Length %d Name %s\n",
        tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, B, tplength);

// Create and print matrix A
makeDblMat(A);
if(N <= 36)
  printDblMat(A, 'A');

// Put chunk_size of A in ts
// Set size of int
tplength = sizeof(int);
// Set name of entry to chunk_size
sprintf(tpname,"chunk_size",0);
printf("Master: Putting chunk_size Length %d Name %s\n",
        tplength, tpname);
// Put entry in tuple space
status = cnf_tsput(tsd, tpname, &chunk_size, tplength);

// Put chunks of A in ts
// Get Ai tuple size
tplength = (2+chunk_size*N) * sizeof(double);
printf("Master: Ai tplength = (%d)\n", tplength);

// Prepare integer array rows for tuple space exchange
if((rows = (double *) malloc(tplength)) == NULL)
  exit(1);
printf("Master: Putting A in problem tuple space\n");

// Build Ai array rows to send to ts
// Set remaining to total number of rows
remaining = N;
// Set start matrix row to zero
```

```
matrix_row = 0;
// Loop until all numbers are sent to workers
while (remaining > 0) {
  // If remaining rows is less than chunk size
  // set number of rows sent to remaining rows
  if (remaining < chunk_size)
    chunk_size = remaining;
  // Subtract rows being sent from remaining rows
  remaining = remaining - chunk_size;
  // Set rows array position to 2
  // Second position (1) is reserved for
  // time returned by worker
  array_pos = 2;
  // Put chunk_size in last index
  rows[0] = chunk_size;
  // Put rows of A matrix in rows array
  for (i=0; i<chunk_size; i++){
    for (j=0; j<N; j++){
      rows[array_pos] = A[matrix_row+i][j];
      if(N <= 36)
        printf(" %g", rows[array_pos]);
      array_pos++;
    }
    if(N <= 36)
      printf("\n");
  }
  // Set entry name to beginning Ai-row
  sprintf(tpname,"A%d",matrix_row);
  printf("Master: Putting chunk_size %d matrix_row %s remaining %d\n",
          chunk_size, tpname, remaining);
  status = cnf_tsput(tsd, tpname, rows, tplength);
  matrix_row += chunk_size;
}

printf("Master: All work has been sent\n");


// Get the result Ci from ts and assemble
// Set received rows to N
remaining = N;
// Initialize worker time
worker_time = 0;
// Loop until all rows are recieved
while(remaining > 0){
  // Set entry name
  strcpy(tpname,"*");
  // Get entry from result tuple space
  tplength = cnf_tsget(res, tpname, rows, 0);
  // Get number rows in this chunk from last index
  chunk_size = rows[0];
  // Get time returned by worker
  worker_time += rows[1];
  // Convert beginning row of entry to an integer
  matrix_row = atoi(tpname);
  printf("Master: Recieved chunk_sizs %d matrix_row %s\n",
         chunk_size, tpname);
```

```
    // Set the position in the array to 2
    array_pos = 2;

    // Assemble the result matrix C
    // Loop through recieved rows
    if(N <= 36)
      printf("Master: Recieved\n");
    for (i= 0; i<chunk_size; i++){
      // Increment rows recieved by decrementing remaining
      remaining--;
      // Loop through row and array elements
      for (j=0; j<N; j++){
        C[matrix_row][j] = rows[array_pos];
        if(N <= 36)
          printf(" %g", C[matrix_row][j]);
        // Increment array position
        array_pos++;
      }
      if(N <= 36)
        printf("\n");
      // Increment row position
      matrix_row++;
    }
  }

  printf("Master: Recieved all work from workers\n");
  printf("Master: C matrix has been assembled\n");

  // Resolve total time
  total_time = wall_clock() - total_time;
  printf("Master: The multiplication took %g seconds total time\n",
         (total_time/1000000.0));

  // Resolve worker time
  printf("Master: The workers used %g seconds of processor time\n",
         (worker_time/1000000.0));

  // Check and print the C matrix
  if(N <= 36)
    printDblMat(C, 'C');
  checkDblIdenMat(C, 'C');

  // Insert termination signal
  // Set length of entry
  tplength = sizeof(double);
  // Set entry value
  i  = -1;
  // Set entry name
  strcpy(tpname, "A-term");
  // Send entry to tuple space
  status = cnf_tsput(tsd, tpname, &i, tplength);

  // Free memory for rows array
  free(rows);

  // Terminate program
```

```
  printf("Master: Terminated\n");
  cnf_term();
}
```

The following is the tuple space “optimized matrix multiplication” worker program implemented by sending work in chunks:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

double Ai[N/2][N];  // A chunk of A matrix
double B[N][N];     // B matrix
double Ci[N/2][N];  // A chunk of C matrix

#include "matrix.h"


// Main function
main(){

  int chunk_size;     // Chunk size
  int i, j, k;        // Matrix indices
  int matrix_row;     // Index of matrix row
  int array_pos;      // Get array position in rows array
  int status;         // Return status for tuple operations
  int res;            // Result tuple space identifier
  int tsd;            // Problem tuple space identifier
  double *rows;       // Rows from A
  double worker_time; // Time to return to master
  int tplength;       // Length of ts entry
  char tpname[20];    // Identifier of ts entry
  char host[128];     // Host machine name

  // Get host machine name
  gethostname(host, sizeof(host));

  // Open tuple spaces
  printf("Worker: Opening tuple spaces\n");
  // Open problem tuple space
  tsd = cnf_open("problem", 0);
  // Open result tuple space
  res = cnf_open("result", 0);
  printf("Worker: Tuple spaces open complete\n");

  // Set tpname to B
  strcpy(tpname,"B");
  // Read matrix B from tuple space
  status = cnf_tsread(tsd, tpname, B, 0);

  // Print matrix B
  if(N <= 36)
    printDblMat(B, 'B');
```

```
// Get chunk_size from master
// Set tpname to chunk_size
strcpy(tpname,"chunk_size");
// Read chunk_size from tuple space
status = cnf_tsread(tsd, tpname, &chunk_size, 0);

// Prepare integer array for tuple space exchanges
tplength = (2+chunk_size*N)*sizeof(double);
if ((rows = (double*)malloc(tplength)) == NULL)
  exit(-1);

// Loop until terminal signal is recieved
while(1){
  // Set entry name to any begins with A
  strcpy(tpname,"A*");
  // Set length of entry
  tplength = cnf_tsget(tsd, tpname, rows, 0);
  // Normal recieve
  if(tplength > 0){
    // Check termination signal
    if (!strcmp(tpname, "A-term")){
      printf("Worker: Recieved the terminal signal\n");
      // Replace the terminal signal in problem ts
      status = cnf_tsput(tsd, tpname, rows, tplength);
      // Free memory for rows
      free(rows);
      // Terminate worker
      printf("Worker: Terminated\n");
      cnf_term();
    }
    // Get number rows in this chunk from last index
    chunk_size = (int)rows[0];
    // Convert beginning row of entry to an integer
    matrix_row = atoi(&tpname[1]);
    printf("Worker: Recieved chunk_size %d matrix_row %d\n",
            chunk_size, matrix_row);

    // Get beginning worker time
    worker_time = wall_clock();

    // For each row in chunk_size
    // Copy rows from rows to Ai
    for(i=0; i<chunk_size; i++)
      for(j=0; j<N; j++){
        Ai[i][j] = rows[i*N+j+2];
        Ci[i][j] = 0;
      }

    // Perform multiplication
    for(i=0; i<chunk_size; i++)
      for(k=0; k<N; k++)
        for(j=0; j<N; j++)
          Ci[i][j] += Ai[i][k]*B[k][j];

    // For each row in chunk_size
    // Copy rows from Ci to rows
```

```
      for(i=0; i<chunk_size; i++){
        for(j=0; j<N; j++)
          rows[i*N+j+2] = Ci[i][j];
      }

      // Put worker time in rows array
      rows[1] = wall_clock() - worker_time;
      // Set tpname to first row number in rows
      sprintf(tpname,"%d", matrix_row);
      printf("Worker: Putting chunk_size %d matrix_row %s\n",
              chunk_size, tpname);
      // Put the result in the result tuple space
      status = cnf_tsput(res, tpname, rows, tplength);
    }

    else{
      printf("Worker: Recieved a zero length entry\n");
      // Free memory for rows
      free(rows);
      // Terminate worker
      printf("Worker Terminated\n");
      cnf_term();
    }
  }
}
```

To run the matrix multiplication distributed application with chunk size of 200 and N = 500 (a 500 x 500 matrix):

1. Set the factor value in the csl file to 200 (as shown below)
2. Make the executables by typing “make SIZE=500” and pressing the enter key.
3. Run the application by typing “prun tupleMat3” and pressing the enter key.

```
configuration: tupleMat3;

m: master = tupleMat3Master
        (factor = 200
        threshold = 1
        debug = 0
        )
   -> f: problem
        (type = TS)
   -> m: worker = tupleMat3Worker
        (type = slave)
   -> f: result
        (type = TS)
   -> m: master;
```

The screen output for the master terminal with Synergy’s initialization and termination output removed should resemble:

```
Master: Opening tuple spaces
Master: Tuple spaces open complete
Master: Processors 2
Master: Chunk size 200
Master: Starting C = A . B
  on 500 x 500 matrices
Master: Putting B Length 2000000 Name B
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Master: Putting chunk_size Length 4 Name chunk_size
Master: Ai tplength = (800016)
Master: Putting A in problem tuple space
Master: Putting chunk_size 200 matrix_row A0 remaining 300
Master: Putting chunk_size 200 matrix_row A200 remaining 100
Worker: Recieved chunk_size 200 matrix_row 200
Master: Putting chunk_size 100 matrix_row A400 remaining 0
Master: All work has been sent
Master: Recieved chunk_sizs 200 matrix_row 0
Worker: Putting chunk_size 200 matrix_row 200
Master: Recieved chunk_sizs 200 matrix_row 200
Master: Recieved chunk_sizs 100 matrix_row 400
Master: Recieved all work from workers
Master: C matrix has been assembled
Master: The multiplication took 4.39389 seconds total time
Master: The workers used 6.23962 seconds of processor time
Master: C is Identity Matrix
Master: Terminated
Worker: Recieved the terminal signal
Worker: Terminated
== (tupleMat3) completed. Elapsed [4] Seconds.
[c615111@owin ~/fpc08 ]>
```

The screen output for the worker terminal with Synergy's initialization and termination output removed should resemble:

```
Worker: Opening tuple spaces
Worker: Tuple spaces open complete
Worker: Recieved chunk_size 200 matrix_row 0
Worker: Putting chunk_size 200 matrix_row 0
Worker: Recieved chunk_size 100 matrix_row 400
Worker: Putting chunk_size 100 matrix_row 400
Worker: Recieved the terminal signal
Worker: Terminated
```

Synergy in the Future

Function and Command Reference

Commands

addhost

This command adds a host into the host file. The command fails if the given host is not Synergy capable. The [-f] option forces the insertion even if the host is not ready. A newly added host automatically becomes "selected".

Syntax:

```
[c615111@owin ~ ]>addhost <hostname> [-f]
```

cds

Checks the status of remote daemons. This command prints all available remote hosts to screen and shows their benchmark, name and availability.

Example:

```
[c615111@owin ~ ]>cds
++ Benchmark (186) ++ (owin) ready.
++ Benchmark (2077) ++ (rancor) ready.
++ Benchmark (2109) ++ (saber) ready.
++ Benchmark (1497) ++ (sarlac) ready.
++ Benchmark (186) ++ (lynox) ready.
[c615111@luke ~ ]>
```

```
[c615111@owin ~ ]>cds
 ????? PMD down (129.32.92.82,ewok)
 ????? CID down (129.32.92.66,luke) (c615111)
 ????? CID down (129.32.92.89,ackbar) (c615111)
 ????? CID down (129.32.92.69,r2d2) (c615111)
[c615111@luke ~ ]>
```

```
[c615111@luke ~ ]>cds
 ????? PMD down (129.32.92.82,ewok)
++ Benchmark (371) ++ (luke) ready.
 ????? CID down (129.32.92.89,ackbar) (c615111)
 ????? CID down (129.32.92.69,r2d2) (c615111)
[c615111@luke ~ ]>
```

chosts

This command allows you to toggle the selected and de-selected status of processors. Only the selected processors will be used for immediate parallel processing. The -v option gives the current Synergy connection status. It requires some extra time.

Syntax:

```
[c615111@owin ~ ]>chosts [-v]
```

Example:

```
         Synergy V3.0 : Host Selection Utility
=Status=No.===IP Address=================Host Name==============Login=F Sys.=
[-----] (  1) #129.32.92.82        ewok                          c615111    none
[-----] (  2) #129.32.92.66        luke                          c615111    none
[-----] (  3) #129.32.92.89        ackbar                        c615111    none
[-----] (  4) #129.32.92.69        r2d2                          c615111    none
[-----] (  5) #129.32.92.87        alliance                      c615111    none
[-----] (  6) #129.32.92.91        anakin                        c615111    none
[-----] (  7) #129.32.92.78        bantha                        c615111    none
[-----] (  8) #129.32.92.74        bobafet                       c615111    none
[-----] (  9) #129.32.92.80        c3p0                          c615111    none
[-----] ( 10) #129.32.92.88        chewbaca                      c615111    none
[-----] ( 11) #129.32.92.86        droids                        c615111    none
[-----] ( 12) #129.32.92.68        emperor                       c615111    none
[-----] ( 13) #129.32.92.77        gredo                         c615111    none
[-----] ( 14) #129.32.92.71        jabba                         c615111    none
[-----] ( 15) #129.32.92.76        jawa                          c615111    none
[-----] ( 16) #129.32.92.83        lando                         c615111    none
[-----] ( 17) #129.32.92.84        leia                          c615111    none
[-----] ( 18) #129.32.92.81        owin                          c615111    none
[-----] ( 19) #129.32.92.70        rancor                        c615111    none
       === Enter s(elect) | d(e-select) | c(ontinue):
```

```
[-----] (  3) #129.32.92.89        ackbar                        c615111    none
[-----] (  4) #129.32.92.69        r2d2                          c615111    none
[-----] (  5) #129.32.92.87        alliance                      c615111    none
[-----] (  6) #129.32.92.91        anakin                        c615111    none
[-----] (  7) #129.32.92.78        bantha                        c615111    none
[-----] (  8) #129.32.92.74        bobafet                       c615111    none
[-----] (  9) #129.32.92.80        c3p0                          c615111    none
[-----] ( 10) #129.32.92.88        chewbaca                      c615111    none
[-----] ( 11) #129.32.92.86        droids                        c615111    none
[-----] ( 12) #129.32.92.68        emperor                       c615111    none
[-----] ( 13) #129.32.92.77        gredo                         c615111    none
[-----] ( 14) #129.32.92.71        jabba                         c615111    none
```

```
[-----] ( 15) #129.32.92.76        jawa                          c615111    none
[-----] ( 16) #129.32.92.83        lando                         c615111    none
[-----] ( 17) #129.32.92.84        leia                          c615111    none
[-----] ( 18) #129.32.92.81        owin                          c615111    none
[-----] ( 19) #129.32.92.70        rancor                        c615111    none
        === Enter s(elect) | d(e-select) | c(ontinue): s
        === Host From (0 to continue) #: 1
                              To      #: 4
         (129.32.92.82 ewok) selected.
         (129.32.92.66 luke) selected.
         (129.32.92.89 ackbar) selected.
         (129.32.92.69 r2d2) selected.
        === Enter s(elect) | d(e-select) | c(ontinue):
```

```
         Synergy V3.0 : Host Selection Utility
=Status=No.===IP Address================Host Name==============Login=F Sys.=
[-----] (  1) 129.32.92.82         ewok                          c615111    none
[-----] (  2) 129.32.92.66         luke                          c615111    none
[-----] (  3) 129.32.92.89         ackbar                        c615111    none
[-----] (  4) 129.32.92.69         r2d2                          c615111    none
[-----] (  5) #129.32.92.87        alliance                      c615111    none
[-----] (  6) #129.32.92.91        anakin                        c615111    none
[-----] (  7) #129.32.92.78        bantha                        c615111    none
[-----] (  8) #129.32.92.74        bobafet                       c615111    none
[-----] (  9) #129.32.92.80        c3p0                          c615111    none
[-----] ( 10) #129.32.92.88        chewbaca                      c615111    none
[-----] ( 11) #129.32.92.86        droids                        c615111    none
[-----] ( 12) #129.32.92.68        emperor                       c615111    none
[-----] ( 13) #129.32.92.77        gredo                         c615111    none
[-----] ( 14) #129.32.92.71        jabba                         c615111    none
[-----] ( 15) #129.32.92.76        jawa                          c615111    none
[-----] ( 16) #129.32.92.83        lando                         c615111    none
[-----] ( 17) #129.32.92.84        leia                          c615111    none
[-----] ( 18) #129.32.92.81        owin                          c615111    none
[-----] ( 19) #129.32.92.70        rancor                        c615111    none
        === Enter s(elect) | d(e-select) | c(ontinue):
```

```
[-----] (  1) 129.32.92.82         ewok                          c615111    none
[-----] (  2) 129.32.92.66         luke                          c615111    none
[-----] (  3) 129.32.92.89         ackbar                        c615111    none
[-----] (  4) 129.32.92.69         r2d2                          c615111    none
[-----] (  5) #129.32.92.87        alliance                      c615111    none
[-----] (  6) #129.32.92.91        anakin                        c615111    none
[-----] (  7) #129.32.92.78        bantha                        c615111    none
[-----] (  8) #129.32.92.74        bobafet                       c615111    none
[-----] (  9) #129.32.92.80        c3p0                          c615111    none
[-----] ( 10) #129.32.92.88        chewbaca                      c615111    none
[-----] ( 11) #129.32.92.86        droids                        c615111    none
[-----] ( 12) #129.32.92.68        emperor                       c615111    none
[-----] ( 13) #129.32.92.77        gredo                         c615111    none
[-----] ( 14) #129.32.92.71        jabba                         c615111    none
[-----] ( 15) #129.32.92.76        jawa                          c615111    none
[-----] ( 16) #129.32.92.83        lando                         c615111    none
```

```
[-----] ( 17) #129.32.92.84        leia                          c615111    none
[-----] ( 18) #129.32.92.81        owin                          c615111    none
[-----] ( 19) #129.32.92.70        rancor                        c615111    none
        === Enter s(elect) | d(e-select) | c(ontinue): d
        === Host From (0 to continue) #: 2
                              To      #: 3
         (luke, #129.32.92.66) de-selected.
         (ackbar, #129.32.92.89) de-selected.
        === Enter s(elect) | d(e-select) | c(ontinue):
```

```
         Synergy V3.0 : Host Selection Utility
=Status=No.===IP Address=================Host Name==============Login=F Sys.=
[-----] (  1) 129.32.92.82         ewok                          c615111    none
[-----] (  2) #129.32.92.66        luke                          c615111    none
[-----] (  3) #129.32.92.89        ackbar                        c615111    none
[-----] (  4) 129.32.92.69         r2d2                          c615111    none
[-----] (  5) #129.32.92.87        alliance                      c615111    none
[-----] (  6) #129.32.92.91        anakin                        c615111    none
[-----] (  7) #129.32.92.78        bantha                        c615111    none
[-----] (  8) #129.32.92.74        bobafet                       c615111    none
[-----] (  9) #129.32.92.80        c3p0                          c615111    none
[-----] ( 10) #129.32.92.88        chewbaca                      c615111    none
[-----] ( 11) #129.32.92.86        droids                        c615111    none
[-----] ( 12) #129.32.92.68        emperor                       c615111    none
[-----] ( 13) #129.32.92.77        gredo                         c615111    none
[-----] ( 14) #129.32.92.71        jabba                         c615111    none
[-----] ( 15) #129.32.92.76        jawa                          c615111    none
[-----] ( 16) #129.32.92.83        lando                         c615111    none
[-----] ( 17) #129.32.92.84        leia                          c615111    none
[-----] ( 18) #129.32.92.81        owin                          c615111    none
[-----] ( 19) #129.32.92.70        rancor                        c615111    none
        === Enter s(elect) | d(e-select) | c(ontinue):
```

cid

Example:

```
[c615111@luke ~ ]>cid &
[1] 23104
[c615111@luke ~ ]> CID HOST NAME (luke)
 Actual CID IP(129.32.92.66)

CID ready.
[c615111@owin ~ ]>
```

```
[c615111@owin ~ ]>cid &
[2] 240
```

```
[c615111@owin ~ ]> CID HOST NAME (owin)
 Actual CID IP(129.32.92.81)

Found an old CID.
Removed an old CID
Reusing cid entry.
CID ready.
[c615111@owin ~ ]>
```

delhost

This command permanently deletes a host from the host file. It fails if the host is Synergy ready. The [-f] option forces the removal.

Syntax:

```
[c615111@owin ~ ]>delhost <hostname> [-f]
```

Example:

dhosts

This command lets you permanently delete more than one host at a time. The -v option will verify the hosts' current Synergy connection status (it takes some extra time).

Syntax:

```
[c615111@owin ~ ]>dhosts [-v]
```

Example:

kds

This command kills all remote daemons. It only kills the daemons started by your own login. It will NOT kill daemons started by others.

pcheck

Utility to check and maintain running parallel programs

Syntax:

```
[c615111@owin ~ ]>pcheck
```

Example:

pmd

Example:

```
[c615111@ewok ~ ]>pmd &
[1] 24172
[c615111@ewok ~ ]>
```

```
[c615111@luke ~ ]>pmd &
[2] 23106
[c615111@luke ~ ]>PMD already running.

[2]    Exit 1                        pmd
[c615111@luke ~ ]>
```

prun

Example:

```
[c615111@owin ~/example01 ]>prun tupleHello1
== Checking Processor Pool:
++ Benchmark (185) ++ (owin) ready.
++ Benchmark (1487) ++ (rancor) ready.
++ Benchmark (1482) ++ (saber) ready.
== Done.
== Parallel Application Console: (owin)
== CONFiguring: (tupleHello1.csl)
== Default directory: (/usr/classes/cis6151/c615111/example01)
++      Automatic program assignment: (worker)->(owin)
++      Automatic slave generation: (worker1)->(rancor)
++      Automatic slave generation: (worker2)->(saber)
++      Automatic program assignment: (master)->(owin)
++      Automatic object assignment: (problem)->(owin) pred(1) succ(3)
++      Automatic object assignment: (result)->(owin) pred(3) succ(1)
== Done.
== Starting Distributed Application Controller ...
Verifying process [|(c615111)|*/tupleHello1Worker
```

```
Verifying process [|(c615111)|*/tupleHello1Worker
Verifying process [|(c615111)|*/tupleHello1Master
Verifying process [|(c615111)|*/tupleHello1Worker
** (tupleHello1.prcd) verified, all components executable.
** (tupleHello1.prcd) started.
== (tupleHello1) completed. Elapsed [5] Seconds.
[c615111@owin ~/example01 ]>
```

sds

This command starts daemons on selected hosts (defined in ~/.sng_hosts).

sfs

Example:

shosts

Example:

Functions

cnf_close(id)

PURPOSE:	Close all internal data structures according to type
PARAMETERS:	int id – identifier of object to be closed
RETURNS:	Nothing

cnf_dget(tpname, tpvalue, tpsize)

PURPOSE:	Destructive read a tuple from a direct tuple space
PARAMETERS:	char *tpname – the name of the object to be read from char *tpvalue – address of receiving buffer int tpsize – ?
RETURNS:	int tpsize – the length of the data read in 8-bit bytes

cnf_dinit()

PURPOSE:	Initializes the tid_list before each scatter operation
PARAMETERS:	None
RETURNS:	1 always

cnf_dput(tsd, tid, tpname, tpvalue, tpsize)

PURPOSE:	Inserts a typle into a direct tuple space
PARAMETERS:	int tsd long tpsize char *tid char *tpname char *tpvalue
RETURNS:	?

cnf_dread(tpname, tpvalue, tpsize)

PURPOSE:	Destructive read a tuple from a direct tuple space
PARAMETERS:	int tpsize; char *tpname;

char *tpvalue;
RETURNS: int tpsize

cnf_dzap()

PURPOSE: Removes all local CID's tuples
PARAMETERS: None
RETURNS: 1 if success or an error code otherwise

cnf_eot(id)

PURPOSE: Marks the end of tasks
PARAMETERS: int id - ?
RETURNS: 1 if success or an error code otherwise

cnf_error(errno)

PURPOSE: Prints to the user the kind of error encountered
PARAMETERS: int errno
RETURNS: 1 always

cnf_fflush(id)

PURPOSE: Flushes a file
PARAMETERS: int id – index into cnf_map to get channel #/ptr
RETURNS: 1 if success or 0 if error

cnf_fgetc(id, buf)

PURPOSE: Read a char from file into buffer
PARAMETERS: int id – index into cnf_map to get channel #/ptr
char *buf; – address of receiving buffer
RETURNS: 0 on EOF otherwise 1

int cnf_fgets(id, buf, bufsiz)

PURPOSE: Read a line from file into buffer
PARAMETERS: int id – index into cnf_map to get channel #/ptr
char *buf – address of receiving buffer
int bufsiz – max size of receiving buffer
RETURNS: 0 if EOF otherwise number of bytes read

cnf_fputc(id, buf)

PURPOSE: Write a char from buffer to file
PARAMETERS: int id – index into cnf_map to get channel #/ptr
char buf – address of receiving buffer
RETURNS: 1 if success or 0 if error

cnf_fputs(id, buf, bufsiz)

PURPOSE: Write a line from buffer to file
PARAMETERS: int id – index into cnf_map to get channel #/ptr
char *buf – address of receiving buffer
int bufsiz – size of buffer
RETURNS: Number of bytes written or 0 if error

cnf_fread(id, buf, bufsiz, nitems)

PURPOSE: Read a 'record' from file into buffer
PARAMETERS: int id – index into cnf_map to get channel #/ptr
char *buf – address of receiving buffer
int bufsiz – max size of receiving buffer
int nitems – number of bufsiz blocks to read
RETURNS: 0 if EOF otherwise number of bytes read

cnf_fseek(id, from, offset)

PURPOSE: Set the reader pointer from "from" to "offset" in a file
PARAMETERS: int id – index into cnf_map to get channel #/ptr
int from
int offset

RETURNS: 1 if success or 0 if error

cnf_fwrite(id, buf, bufsiz, nitems)

PURPOSE: Write a 'record' from buffer into file
PARAMETERS: int id – index into cnf_map to get channel #/ptr
char *buf – address of receiving buffer
int bufsiz – max size of receiving buffer
int nitems – number of bufsiz blocks to write
RETURNS: Number of bytes written or an error code on error

cnf_getarg(idx)

PURPOSE: Returns the runtime argument by index
PARAMETERS: int idx – the index
RETURNS: char * (idx'th argument)

cnf_getf()

PURPOSE: Returns the factor value for loop scheduling
PARAMETERS: None
RETURNS: f value (0..100] integer

cnf_getP()

PURPOSE: Returns the number of parallel workers
PARAMETERS: None
RETURNS: P value [1..N] integer

cnf_gett()

PURPOSE: Returns the threshold value for loop scheduling
PARAMETERS: None
RETURNS: t value [1..N) integer

cnf_gts(tsd)

PURPOSE:	Get all tid's processor assignments in one shot
PARAMETERS:	int tsd - ?
RETURNS:	1 if success, 0 if no memory or an error code otherwise

cnf_init()

PURPOSE:	Initializes sng_map_hd and sng_map using either the init file or direct transmission from DAC. The init file's name is constructed from the value of the logical name CNF_MODULE suffixed with ".ini".
PARAMETERS:	None
RETURNS:	Nothing if successful or an error code otherwise

cnf_open(local_name, mode)

PURPOSE:	Lookup a pipe or tuple space object in sng_map structure, open a channel to the physical address for that ref_name
PARAMETERS:	char *local_name – local_name to find in cnf_map char *mode – open modes: r,w,a,r+,w+,a+. Only for FILEs
RETURNS:	int chan – an integer handle, if successful or an error code otherwise. This is used like a usual Unix file handle.

cnf_print_map()

PURPOSE:	?
PARAMETERS:	None
RETURNS:	Nothing

cnf_read(id, buf, bufsiz)

PURPOSE:	read a 'record' from file or pipe into buffer (starting at address buff).
PARAMETERS:	int id – index into cnf_map to get channel #/ptr int bufsiz – max size of receiving buffer char *buf – address of receiving buffer

RETURNS: 0 on EOF otherwise number of bytes read

cnf_rmall(id)

PURPOSE: Destroy all tuples in a named tuple space
PARAMETERS: int id - ?
RETURNS: 0 if successful or an error code otherwise

cnf_sot(id)

PURPOSE: Marks the start of scantering of tasks
PARAMETERS: int id
RETURNS: 1 if successful or an error code otherwise

cnf_spzap(tsd)

PURPOSE: Removes all "retrieve" entries in TSH
PARAMETERS: int tsd - ?
RETURNS: 1 if successful or an error code otherwise

cnf_term()

PURPOSE: Called before image return to clean things up. Closes any files left open.
PARAMETERS: None
RETURNS: Nothing

cnf_tget(tpname, tpvalue, tpsize)

PURPOSE: Destructive read a tuple from a named tuple space
PARAMETERS: int tpsize -
char *tpname -
char *tpvalue -
RETURNS: int tpsize – the size of the tuple received if successful or an error code otherwise

cnf_tsput(tpname, tpvalue, tpsize)

PURPOSE:	Inserts a tuple into a named tuple space
PARAMETERS:	int tpsize - char *tpname - char *tpvalue -
RETURNS:	? on success or an error code otherwise

cnf_tsread(tpname, tpvalue, tpsize)

PURPOSE:	Read a tuple from a named tuple space
PARAMETERS:	int tpsize - char *tpname - char *tpvalue -
RETURNS:	int tpsize – the size of the tuple received if successful or an error code otherwise

cnf_tsget(id, tpname, tpvalue, tpsize)

PURPOSE:	Destructive read a tuple from a named tuple space
PARAMETERS:	int id - int tpsize - char *tpname - char *tpvalue -
RETURNS:	int tpsize – the size of the tuple received if successful or an error code otherwise

cnf_tsput(id, tpname, tpvalue, tpsize)

PURPOSE:	`Inserts a tuple into a named tuple space`
PARAMETERS:	int id - int tpsize - char *tpname - char *tpvalue -
RETURNS:	? on success or an error code otherwise

cnf_tsread(id, tpname, tpvalue, tpsize)

PURPOSE:	Read a tuple from a named tuple space
PARAMETERS:	int id - int tpsize - char *tpname - char *tpvalue -
RETURNS:	int tpsize – the size of the tuple received if successful or an error code otherwise

cnf_write(id, buf, bytes)

PURPOSE:	Send a 'record' to file (or mailbox or decnet channel) from buffer (starting at address buff). bytes is the number of bytes to send. id is the index into cnf_map global data structure where the actual channel number or file pointer is stored.
PARAMETERS:	int id – index into cnf_map for channel #/ptr int bytes – number of bytes to send/write char buf[] – address of message to send
RETURNS:	1 if successful or an error code otherwise

cnf_xdr_fgets(id, buf, bufsize, e_type)

PURPOSE:	Read the external data representation of a line from file into buffer (starting at address xdr_buff) and translates it to C language.
PARAMETERS:	int id – The index into cnf_map global data structure where the actual channel number or file pointer is stored char *buf - int bufsize – the number of bytes to read int e_type -
RETURNS:	0 on EOF or number of bytes read on success otherwise an error code on error

cnf_xdr_fputs(id, buf, bufsize, e_type)

PURPOSE:	Translates a line to it's external data representation and sends it to file from buffer (starting at address xdr_buff). .

PARAMETERS: int id – The index into cnf_map global data structure where the actual channel number or file pointer is stored
char *buf -
int bufsize – the number of bytes to send
int e_type -
RETURNS: int status - number of bytes written, 0 if error writing or an error code otherwise

cnf_xdr_fread(id, buf, bufsize, nitems, e_type)

PURPOSE: Read the external data representation of a 'record' from file into buffer (starting at address xdr_buff) and translates it to C language.
PARAMETERS: int id – The index into cnf_map global data structure where the actual channel number or file pointer is stored
char *buf -
int bufsize – the number of bytes to read
int nitems -
int e_type -
RETURNS: int status - number of bytes read, 0 if error writing or an error code otherwise

cnf_xdr_fwrite(id, buf, bufsize, nitems, e_type)

PURPOSE: Translates a 'record` to it's external data representation and sends it to file from buffer (starting at address xdr_buff).
PARAMETERS: int id – The index into cnf_map global data structure where the actual channel number or file pointer is stored
char *buf -
int bufsize – the number of bytes to send
int nitems -
int e_type -
RETURNS: Number of bytes written or an error code or -1 on error

cnf_xdr_read(id, buf, bufsize, e_type)

PURPOSE: Read the external data representation of a 'record' from file or pipe into buffer (starting at address xdr_buff) and translates it to C language.

PARAMETERS: int id – The index into cnf_map global data structure where the actual channel number or file pointer is stored
char *buf -
int bufsize – the number of bytes to read
int e_type -
RETURNS: int status - number of bytes read, 0 if error writing or an error code otherwise

cnf_xdr_tsget(tsh, tp_name, tuple, tp_len, e_type)

PURPOSE: Destructive reads the external data representation of a tuple from a named tuple space and Translates it to C language.
PARAMETERS: int tsh
char *tp_name
char *tuple
int tp_len
int e_type
RETURNS: int status - the size of the tuple received if successful, 0 if it is an asynchronous read or –1 on error

cnf_xdr_tsput(tsh, tp_name, tuple, tp_len, e_type)

PURPOSE: Translates a tuple to it's external data representation and inserts it into a named tuple space
PARAMETERS: int tsh
char *tp_name
char *tuple
int tp_len
int e_type
RETURNS: int status - ? on success or an error code otherwise

cnf_xdr_tsread(tsh, tp_name, tuple, tp_len, e_type)

PURPOSE: Reads the external data representation of a tuple from a named tuple space and translates it to C language.
PARAMETERS: int tsh
char *tp_name
char *tuple

int tp_len
int e_type
RETURNS: int status - number of bytes read, 0 if error writing or an error code or –1 on error

cnf_xdr_write(id, buf, bufsize, e_type)

PURPOSE: Translates a 'record` to it's external data representation and sends it to file (or mailbox or decnet channel) from buffer (starting at address xdr_buff).
PARAMETERS: int id – The index into cnf_map global data structure where the actual channel number or file pointer is stored
char *buf -
int bufsize – the number of bytes to send
int e_type -
RETURNS: 1 if successful or an error code or –1 on error

PURPOSE:
PARAMETERS:
RETURNS:

Error Codes

TSH_ER_NOERROR	Normal operation - No error at all
TSH_ER_INSTALL	Error: Tuple Space daemon could not be started
TSH_ER_NOTUPLE	Error: Could not find such tuple
TSH_ER_NOMEM	Error: Tuple space daemon out of memory
TSH_ER_OVERRT	Warning: Tuple was overwritten

References

[i] Information on tally sticks found at members.fortunecity.com
[ii] Information on abacus found at http://www.maxmon.com
[iii] Jill Britton, Department of Mathematics, Camosun College, 3100 Foul Bay Road, Victoria, BC, Canada, V8P 5J2. Web Page: http://ccins.camosun.bc.ca/~jbritton/jberatosthenes.htm
[iv] http://encyclopedia.thefreedictionary.com/
[v] http://www.thocp.net/hardware/pascaline.htm
[vi] http://www.ox.compsoc.net/~swhite/history/timelines.html
[vii] http://miami.int.gu.edu.au/dbs/1010/lectures/lecture4/Ifrah-pp121-133.html
[viii] http://www.agnesscott.edu/lriddle/women/love.htm
[ix] http://www.kerryr.net/pioneers/boole.htm
[x] http://knight.city.ba.k12.md.us/faculty/ss/samuelmorse.htm
[xi] http://history.acusd.edu/gen/recording/bell-evolution.html
[xii] http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Hollerith.html - Article by: J J O'Connor and E F Robertson
[xiii] http://www.marconi.com/html/about/marconihistory.htm
[xiv] http://www.radio-electronics.com/info/radio_history/gtnames/fleming.html
[xv] http://www.epemag.com/zuse/
[xvi] http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Aiken.html
[xvii] http://www.research.att.com/~njas/doc/shannonbio.html
[xviii] http://www.kerryr.net/pioneers/stibitz.htm
[xix] http://plato.stanford.edu/entries/turing/
[xx] http://ei.cs.vt.edu/~history/do_Atanasoff.html
[xxi] http://www.library.upenn.edu/exhibits/rbm/mauchly/jwmintro.html
[xxii] http://ftp.arl.mil/~mike/comphist/61ordnance/chap3.html
[xxiii] http://en.wikipedia.org/wiki/MIT_Whirlwind
[xxiv] http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/statistics.html
[xxv] http://www.computer50.org/mark1/MM1.html
[xxvi] http://www.awc-hq.org/lovelace/1997.htm
[xxvii] http://www.cs.yale.edu/homes/tap/Files/hopper-story.html
[xxviii] http://inventors.about.com/library/weekly/aa061698.htm
[xxix] http://csdl.computer.org/comp/mags/an/2004/02/a2034abs.htm
[xxx] http://www.cc.gatech.edu/gvu/people/randy.carpenter/folklore/v3n1.html
[xxxi] http://en.wikipedia.org/wiki/Defense_Advanced_Research_Projects_Agency
[xxxii] http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html#history
[xxxiii] http://inventors.about.com/library/weekly/aa080498.htm
[xxxiv] http://www.nersc.gov/~deboni/Computer.history/LARC.Cole.html
[xxxv] http://www.smartcomputing.com/editorial/dictionary/detail.asp?guid=&searchtype=1&DicID=16502&RefType=Encyclopedia
[xxxvi] http://en.wikipedia.org/wiki/CTSS
[xxxvii] http://www.ukuug.org/events/linux2001/papers/html/DAspinall.html
[xxxviii] http://www.fys.ruu.nl/~bergmann/history.html
[xxxix] http://www.engin.umd.umich.edu/CIS/course.des/cis400/pl1/pl1.html
[xl] http://www.afrlhorizons.com/Briefs/Mar02/OSR0103.html
[xli] http://www.smalltalk.org/alankay.html
[xlii] http://www.faqs.org/faqs/dec-faq/pdp8/

[xliii] http://bugclub.org/beginners/languages/pascal.html
[xliv] http://en.wikipedia.org/wiki/Edsger_Dijkstra
[xlv] http://www.campusprogram.com/reference/en/wikipedia/s/so/software_engineering.html
[xlvi] http://en.wikipedia.org/wiki/UNIX
[xlvii] http://en.wikipedia.org/wiki/RS-232
[xlviii] http://inventors.about.com/library/weekly/aa092998.htm
[xlix] http://bugclub.org/beginners/processors/Intel-8086.html
[l] http://bugclub.org/beginners/processors/Intel-80186.html
[li] http://www.pcguide.com/ref/cpu/char/mfg.htm
[lii] http://members.fortunecity.com/pcmuseum/dos.htm
[liii] http://www.cs.uiuc.edu/news/alumni/fa98/chen.html
[liv] http://www.webmythology.com/VAXhistory.htm
[lv] http://en.wikipedia.org/wiki/Motorola_68000
[lvi] http://en.wikipedia.org/wiki/INMOS_Transputer
[lvii] http://csep1.phy.ornl.gov/ca/node11.html
[lviii] PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing; Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam; MIT Press, Scientific and Engineering Computation; Janusz Kowalik, Editor; Copyright 1994 Massachusetts Institute of Technology. The book can be viewed at: http://www.netlib.org/pvm3/book/pvm-book.html
[lix] Linda Users Guide and Reference Manual, Manual Version 6.2; Copyright © 1989-1994, SCIENTIFIC Computing Associates, Inc. All rights reserved.
[lx] http://people.hofstra.edu/faculty/Stefan_Waner/RealWorld/logic/logicintro.html
[lxi] Garson, James, "Modal Logic", *The Stanford Encyclopedia of Philosophy (Winter 2003 Edition)*, Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/win2003/entries/logic-modal/>.
[lxii] Galton, Antony, "Temporal Logic", *The Stanford Encyclopedia of Philosophy (Winter 2003 Edition),* Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/win2003/entries/logic-temporal/>.
[lxiii] Reevaluating Amdahl's Law; John L. Gustafson; Sandia National Laboratories; 1988.
[lxiv] Reevaluating Amdahl's Law and Gustafson's Law; Yuan Shi; Temple University; October 1996.
[lxv] Synergy Manual