Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Materials & Chemical Sciences Division

QUANTUM CHEMICAL METHODS FOR MASSIVELY PARALLEL COMPUTERS

M.E. Colvin
(Ph.D. Thesis)

November 1986

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

Quantum Chemical Methods for Massively Parallel Computers

Michael Eric Colvin

Ph.D. Thesis

Department of Chemistry
University of California, Berkeley
and
Materials and Chemical Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

November, 1986

Quantum Chemical Methods for Massively Parallel Computers

Michael Eric Colvin

Department of Chemistry
University of California
Berkeley, California 94720

ABSTRACT

For many years it has been recognized that fundamental physical constraints such as the speed of light will limit the ultimate speed of single processor computers to less than about three billion floating point operations per second. This limitation is becoming increasingly restrictive as commercially available machines are now within an order of magnitude of this asymptotic limit. A natural way to avoid this limit is to harness together many processors to work on a single computation problem. In principle the net processing speed of such a system is limited only by the number of processors linked together.

The usefulness of potentially unlimited processing speed to a computationally intensive field such as quantum chemistry is obvious. Moreover, if these methods are to be applied to significantly larger chemical systems, parallel schemes will have to be employed. The work described in this thesis represents a first step toward the development of a general system of parallel quantum chemistry programs. Parallel algorithms have been developed for several important quantum chemical techniques: integral evaluation, self consistent field calculation, integral transformation and second order Moller Plesset energy calculation. These algorithms have been implemented and

tested on a 32 processor Intel Hypercube.

The benchmark calculations indicate very good parallel performance for all of these algorithms. The most computationally complex step, the two electron integral transformation, exhibits nearly perfect parallel speedup. That is, when utilizing n processors, the execution speed is increased by a factor of n over the single processor implementation.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my research director, Professor H.F. Schaefer III for providing me with a stimulating research environment and allowing me the freedom to pursue this and other wide-ranging research projects. I would also like to thank the Schaefer group for help and encouragement during my four years at Berkeley. Special thanks go to Curtis Janssen who developed the communication routines for the Sun workstations and shared my enthusiasm for the potential of parallel processors.

I greatly appreciate the help of Dr. Robert A. Whiteside at Sandia National Laboratories who has played an indispensable role in the work described in this thesis. Without his guidance and expert advice though all stages of this project it could not have been accomplished. I would also like to thank Dr. J. Steven Binkley for providing me generous access to Sandia's Hypercube.

Finally, I would like to thank Judy Yang who has given me support and encouragement throughout my graduate career. Her love and companionship have been an essential source of strength to me during the past four years. Thanks.

This work has been supported by both the U.S. National Science Foundation grant CHE-8218785, and by the Director, Office of Energy Research, Chemical Sciences Division of the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

Table of Contents

Chapter I: Introduction to Parallel Computers and Molecular Quantum Mechanics

INTRODUCTION

In 1946 shortly after the development of the first electronic computer John Von Neumann was asked to consult the Office of Naval Research on whether enough applications could be found for high speed computing devices to justify further funding. Von Neumann concluded:

> It is, furthermore, quite clear that there are many problems where the length or the complexity of the problem is not sufficient to justify electronic speeds. It is important, however, to point out that there are plenty of problems which justify this speed, and, furthermore, the chances are that if these speeds become available, we will come to discover more and more how numerous these problems are.[1]

In the forty years since Von Neumann made this comment computers have had a revolutionary impact on nearly all aspects of human endeavor. Today computers are essential tools in many fields including business, medicine, publishing and manufacturing. However no field has been affected more dramatically than scientific research. Computers are a ubiquitous presence in the laboratory where they control instruments and collect and analyze data. Yet the most important function of computers in science is that which Von Neumann originally predicted: as a tool for the accurate theoretical modeling of physical systems.

The potential of such computational models was quickly recognized by scientists and a number of significant research projects were carried out on the earliest available computers.[2,3] In the subsequent decades scientific computation has had a history of impressive achievements and spawned the creation of many new fields of

theoretical research. A large factor in the increasing accuracy and utility of of scientific computation has been the phenomenal growth of computer performance. In the past forty years the instruction speed and memory capacity of computers have increased by nearly six orders of magnitude.

Unfortunately single processor (or serial) performance will not continue to improve at this rate. Despite a thirty year history of a factor of ten increase in instruction speeds every 5-7 years, the past decade has seen only a three fold increase in the clock rates of high-end supercomputers.[4] Moreover, fundamental physical constraints such as the speed of light limit single processor computational speeds to less than about three billion floating point operations per second[5] which is less than an order of magnitude faster than currently available supercomputers. Such a limitation would seriously impair many fields where the size and accuracy of the systems modeled are severely constrained by available computer speeds.

A possible solution to this problem is to harness together many processors to work on a single computational problem. In principle the net processing speed of such a system is limited only by the number of processors linked together. The concept of so called parallel processing computers has been around since the earliest days of automatic computing, and a number of experimental parallel computers were proposed and built in the 1960's and 70's.[6,7,8] Yet, parallel computers have become commercially available only in the last few years with the development of inexpensive and reliable processing elements. Currently there are more than a dozen commercially announced parallel computers ranging from two processor machines operating at less than a megaflop (million floating point operations per second)[9] to 4000 processor machines operating at nearly a hundred gigaflops (billion floating point operations a second).[10]

The potential capabilities of parallel computers have stimulated tremendous

interest in the computer science community (illustrated by the fact that a 1983 bibliography on parallel computers contained a total of 5161 entries).[11] Despite this great enthusiasm among computer scientists, there has been relatively little interest from the physical sciences. This lack of interest is due to the experimental nature of most of the available parallel computers and the difficulty of programming efficiently such machines. Nevertheless there have been a few significant efforts worthy of note, all the result of collaboration between physical scientists and computer scientists.

One of the earliest such efforts was carried out at Carnegie-Mellon University and involved the development of monte carlo and molecular dynamics algorithms for the 50 processor CM* computer.[12] Two large parallel computing projects are currently underway at IBM and Caltech. The IBM project is aimed at developing quantum chemistry and molecular dynamics algorithms for parallel computers involving relatively small numbers of high-end processors.[13] The Caltech project involves the application of very large arrays of microprocessors to a wide variety of problems in the physical and biological sciences.[14] In the near future the number and magnitude of parallel scientific computation projects will certainly grow, as parallel hardware and expertise become more widespread and the need for faster computers more acutely felt.

COMPUTERS AND MOLECULAR QUANTUM MECHANICS

With the development of quantum mechanics in the 1920's it became possible in principle to predict theoretically any chemical property. Such a prediction requires solving the molecular Schrodinger equation, a partial differential equation of the elliptic type which is insoluble for all but trivial cases. Hence, for thirty years theoretical chemists were limited to very small systems (such as helium atom[15] or

hydrogen molecule[16]) or to very simplified models of large compounds.[17] The advent of the electronic computer rapidly expanded the horizons of the quantum chemist. In 1956 Boys reported the first quantum chemical calculation carried out entirely on an electronic computer[3] and only three years later Robert Mulliken predicted: "colossal rewards lie ahead from large scale quantum mechanical calculations of the structure of matter."[18] Since 1959 great strides have been made in this direction. Theoretically predicted molecular properties are used routinely to aid in the interpretation of experimental results, and there have been many instances where discrepancies between theory and experiment have been resolved in favor of theory.[19]

Despite this history of successes, Mulliken's vision of obtaining " the electronic eigenfunction and energy of every major type of molecule" has not been realized. The problem is not that there are inherent limitations in the theoretical methods; these methods could, in principle, be applied to macromolecules or solids. Instead the limitation is due to the inherent complexity and associated computational cost of solving the molecular Schrodinger equation. Even the fastest available supercomputers are not sufficiently fast to allow the study of molecules with more than a few dozen atoms. Hence, the limitations on single processor performance discussed in the last section will seriously constrain the size of systems amenable to study by quantum chemical methods. If quantum chemical calculations are to be carried out on significantly larger systems, parallel computers will have to be used.

The conversion of standard quantum chemical methods to parallel computers will not be an simple task. Most of these methods require large data sets and involve tightly coupled iterative algorithms not easily broken down into concurrent procedures. Moreover, the quantum chemical procedures have been optimized to make efficient use of single processor computers, further complicating the conver-

sion to parallel machines.

A research group a IBM under the direction of Enrico Clementi has successfully implemented a parallel scheme for doing SCF calculations [20] and are currently developing parallel algorithms for other quantum chemical techniques.[21] However, these algorithms are designed for parallel computers having ten or fewer very large scale processors. In order to gain the very dramatic performance increases promised by parallel computers, algorithms must be designed to run efficiently on arbitrarily large arrays of processors. The work described in this thesis represents a first step towards this goal. This work involved the design and implementation of parallel algorithms for several major quantum chemical procedures. These algorithms were implemented on a 32 processor Intel Hypercube, a prototype of the much larger parallel computers that will be available in the next few years.

The remainder of this chapter is divided into three sections. The first is a description of the methods and computational requirements of molecular quantum mechanics. Following this is a brief overview of the types of parallel computers now available and a description of the Intel Hypercube. The third section is a discussion of general principles for parallelizing scientific programs. Chapter two contains a detailed description of the parallel algorithms, and the third chapter gives the benchmark results and an analysis of the algorithm efficiencies. The final chapter discusses quantum chemical methods and computer technologies that seem promising for the future.

METHODS OF MOLECULAR QUANTUM MECHANICS

The goal of molecular quantum mechanics is to calculate from first principles the chemical properties of molecular systems. This amounts to solving the molecular Schrodinger equation to determine the wavefunction of the molecule of interest.

From this wavefunction all observable properties of the molecule can be calculated.

The molecular Schrodinger equation cannot be solved exactly except for the simplest systems, so that a series of approximations must be made. For most chemical applications, relativistic effects are sufficiently small that they can be ignored.[22] Also, terms coupling nuclear motion to the electronic structure can be omitted from the molecular Hamiltonian (Born-Oppenheimer approximation[23]) so that the problem reduces to determining the electronic wavefunction for a fixed nuclear framework.

In most quantum chemical calculations the electronic wavefunction is represented by an antisymmetrized product of molecular orbitals (MO's) known as a Slater determinant.

$$\Psi = \frac{1}{\sqrt{n!}}\det(\phi_1\ \phi_2\ \cdots\ \phi_n) \tag{I.1}$$

These molecular orbitals are approximated as linear combinations of atomic orbitals (AO's)

$$\phi_i = \sum_{\mu} C_{\mu i}\chi_{\mu} \tag{I.2}$$

The matrix **C** relating the molecular orbitals to the atomic orbital basis functions is determined by the iterative self-consistent field (SCF)[24] procedure. The atomic orbital basis functions are represented by analytic functions having the form of linear combinations of gaussian functions:

$$\chi_{\mu} = \sum_{i} d_i\, x^{l_i} y^{m_i} z^{n_i} e^{-\alpha_i r^2} \tag{I.3}$$

The contraction coefficients d are fixed before the computations begins. Typically a quantum chemical calculation involves the use of 50-100 of these basis functions.

The SCF procedure is usually carried out in two steps. The first is the numeri-

cal calculation of integrals over the basis functions. The most time consuming of these integrals has the form:

$$(\mu\nu|\lambda\sigma) = \iint \phi_\mu(1)\phi_\nu(1)\frac{1}{r_{12}}\phi_\lambda(2)\phi_\sigma(2)d\tau_1 d\tau_2 \qquad \text{(I.4)}$$

These integrals have an inherent 8-fold symmetry with respect to the permutation of their indices:

$$(\mu\nu|\lambda\sigma) = (\nu\mu|\lambda\sigma) = (\lambda\sigma|\mu\nu)$$

For a system with n basis functions there are $n^4/8$ of these integrals, hence the computational complexity of this step is $O(n^4)$. The second step is the calculation of the self consistent field energy and molecular orbitals. This iterative procedure requires the repeated construction and diagonalization of a Hamiltonian matrix from the pre-calculated integral list. This Hamiltonian is called the Fock operator and has the form:

$$F_{\mu\nu} = H_{\mu\nu} + \sum_{\lambda\sigma} P_{\lambda\sigma}\left[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\lambda|\sigma\nu)\right] \qquad \text{(I.5)}$$

where H is the sum of the kinetic energy and nuclear attraction integrals, and P is the density matrix formed from the SCF vector C. The complexity of this step is also $O(n^4)$ since the integral list must be processed each iteration. An alternative approach is used when there is insufficient memory to store the two-electron integrals. In this strategy, known as direct SCF, the integral list is recalculated every SCF iteration.

Although many properties calculated form the SCF wavefunction are quantitatively accurate[24] , it is often necessary to go beyond this first order approximation. There are a variety of different methods for extending the accuracy of the SCF wavefunction. The more widely used methods include: configuration interaction[25], Moller Plesset perturbation theory[26], and coupled-cluster methods[27]. These

techniques have the common feature that they all require the AO integral list to be transformed into the MO basis. This transformation has the form:

$$(ij|kl) = \sum_{\mu}\sum_{\nu}\sum_{\lambda}\sum_{\sigma} C_{\mu i} C_{\nu j} C_{\lambda k} C_{\sigma l} (\mu\nu|\lambda\sigma) \tag{I.6}$$

where i,j,k,l, are MO indices, μ,ν,λ,σ, are AO indices, and C is the MO coefficient matrix. As written above, this transformation has complexity $O(n^8)$, but it can be rearranged into a series of partial transformations so that it has complexity $O(n^5)$.[28]

The two most widely used post-SCF techniques are configuration interaction (CI) and Moller-Plesset perturbation theory (MPPT). Both methods have strengths and weaknesses so that they can be used in complementary roles.

Moller-Plesset perturbation theory is standard perturbation theory approach where the zeroth-order Hamiltonian is the Hartree-Fock wavefunction. The perturbation term is the difference between this approximate Hamiltonian and the full molecular Hamiltonian. When the perturbation expansion is carried out corrections to the SCF energy and wavefunction to various orders are obtained. The energy expressions have the form of sums of products of transformed integrals and SCF orbital energies. For example the second order energy correction is:

$$E_0^{(2)} = \frac{1}{4} \sum_{abrs} \frac{|\langle ab||rs\rangle|^2}{\varepsilon_a+\varepsilon_b-\varepsilon_r-\varepsilon_s} \tag{I.7}$$

where ε_a is the SCF energy of orbital a and $\langle ab||rs\rangle$ are the so called super-integrals formed from the transformed integrals as follows:

$$\langle ab||cd\rangle = (ac|bd)-(ad|bc) \tag{I.8}$$

The energy corrections are routinely calculated to fourth order in quantum chemical calculations.

In the configuration interaction technique the molecular wavefunction is approximated as a linear combination of Slater determinants. In general the

wavefunction of any molecular system can be written as

$$|\Phi\rangle = c_0|\Psi_0\rangle + \sum_{ra} c_a^r|\Psi_a^r\rangle + \sum_{\substack{a<b \\ r<s}} c_{ab}^{rs}|\Psi_{ab}^{rs}\rangle + \cdots \tag{I.9}$$

where $|\Psi_0\rangle$ is the Hartree-Fock wavefunction and $|\Psi_a^r\rangle$ represents the ground state wavefunction with an electron excited from the occupied orbital a to virtual orbital r.

The CI energies and wavefunctions are the eigenvalues and eigenvectors of the CI Hamiltonian matrix:

$$\langle\Phi|H|\Phi\rangle \tag{I.10}$$

where H is the molecular Hamiltonian and $|\Phi\rangle$ is the wavefunction given above. Since the number of terms in the CI wavefunction grows exponentially in the number of basis functions and electrons, the expansion is usually truncated to some preset excitation level. The most often used CI wavefunction includes only single and double excitations, which for most systems recovers more than 90% of the correlation energy.[24]

Many interesting chemical properties depend not on the total molecular energies (calculated by the methods just described), but instead depend on the derivative of this energy with respect to nuclear coordinates or some external field. Although these derivatives can be approximated by finite difference techniques from the energy calculations, in the past decade more accurate and efficient methods have been developed. These methods involve explicitly differentiating the energy expressions and then evaluating these exact expressions. These so-called "analytic" derivative methods have been implemented for the first and second derivatives (with respect to nuclear coordinates) for SCF[29], CI[30], and MPPT[31] wavefunctions.

This thesis describes algorithms and implementation results for the integral

evaluation, SCF energy calculation, integral transformation, and second order Moller-Plesset energy calculation. Additionally, an algorithm is described for a parallel configuration interaction program.

Introduction to Parallel Processors

A 1985 survey by the Parallel Processing Research Council found more than 50 ongoing projects developing new parallel computer architectures.[32] Although these computers all involve the interconnection of many processors, the nature of the individual processors and how they are linked together vary greatly form design to design. Despite the vast number of proposed designs it is possible to classify nearly all parallel computers into a relatively small number of categories.

One broad categorization is made on the basis of whether the individual processors carry out the same or different instruction streams. One design is labeled single instruction multiple data stream (SIMD) and refers to computers where a number of processing elements carry out the same instructions in lock-step on parallel streams of data. SIMD machines are well suited for application such as numerical solution of differential equations where the identical numerical computation is carried out on each distinct grid point. The classic example of a SIMD machine is the Illiac IV[7] which contained an 8x8 grid of processors. A recent and more exotic SIMD computer is the "Connection Machine"[33] which contains 65,000 one bit processors and was designed for both scientific and artificial intelligence applications. The limitation of SIMD computers is that they can be used efficiently only when the computational problems well match the the uniform structure of the machines.

The alternate category is the multiple instruction multiple data stream (MIMD) computers. MIMD describes any linked collection of processing elements which are not constrained to carry out identical instruction streams (as are SIMD machines).

To perform useful computation the processors have to be able to communicate, and it is this communication scheme which defines the subcategories of MIMD computers.

The two most common communication strategies are for the processors to share a common pool of memory (shared memory) or to communicate explicitly by sending and receiving message packets (distributed memory). Many of the proposed and currently available MIMD computers use shared memory. In most shared memory computers any processor can access any value in the common memory. This makes programming shared memory computers relatively straight forward and for this reason most research on automatic parallelizing compilers (i.e. compilers which convert serial programs to parallel) is targeting shared memory computers.

There is, however, a drawback to the shared memory architecture. As the number of processors becomes large the communication with the shared memory can become a bottleneck. It is not yet known how restrictive this bottleneck will prove to be, but most such machines have less than 20 processing elements. A possible way to void the limitation is to have a hierarchy of common memories in which small clusters of processors share a common memory and each cluster has a single channel to a global shared memory.[34] Of course, the introduction of such a hierarchy is at the cost of programming simplicity which was the prime motivation for using the shared memory architecture.

Distributed memory architectures have the advantage that there are in principle no hardware bottlenecks since communication between one pair of processors is independent of communication between another pair. The disadvantage of distributed memory systems is that it is more difficult to program since passing data requires synchronizing both the sending and receiving processors.

An important design parameter of distributed memory computers is the connec-

tion topology of the processing elements--that is, the way in which the processors are linked by communication channels. A large variety of communication topologies have been studied. These are usually regular grids in one or more dimensions (where the grid's vertices represent processing elements and the edges represent communication channels). The conclusion of these studies has been that the optimal processor connectivity is dictated by the computational problem at hand.[8] Hence, one option is to build a special purpose processor for each problem to be solved. A more practical alternative is to use a complex connectivity that has a large number of useful sub-topologies.

An example of the second alternative are the hypercube architectures. In a hypercube the processor connectivity is that of a n-dimensional boolean hypercube. A hypercube of dimension n has 2^n processors each connected to n neighbors. Hypercubes of dimension 1, 2, 3, and 4 are shown in figure 1. This connection topology allows the embedding of regular meshes of lower dimension than the hypercube using a subset of the connections of the hypercube. Such mappings are easily carried out using a technique known as Gray coding.[35]

The goal of the work described in this thesis is to demonstrate the feasibility of very dramatic speedups through the use of parallel computers. Such performance increases require the use of very large numbers of processors which are currently available only in distributed memory computers. For this reason the algorithms presented were designed with regard to the advantages and limitations of distributed memory machines.

The implementation of the programs was carried out on an Intel IPSC parallel computer (hereafter referred to as the Hypercube) located at Sandia National Laboratory in Livermore, California. The Hypercube is a 32 processor distributed memory computer with hypercube connectivity. In addition to these 32 "node" processors,

there is a host processor with a communication channel to each node processor. Each processing node contains an INTEL 80286 CPU and an 80287 floating point processor as well as 512 Kbytes of RAM. The host is an identical processing element with 2 Mbytes of RAM and access to a 20 Mbyte hard disk. The communication links are ethernet channels with a maximum bandwidth of 10 megabits per second.

A series of benchmark calculations were run on the Hypercube to determine its overall performance. These indicate that if all 32 processors are running at 100% efficiency, the Hypercube operates at 0.62 million floating point operations per second. The node to node communication rate to be 225 kbits per second for nearest neighbor nodes. Although it is possible to send messages between non-nearest neighbor nodes, this causes a significant degradation in communication speed. Taken together, these results indicate that a node can carry out 65 floating point operations in the time it can transmit 1 Kbyte of data to the neighboring processor. The details of these benchmark results are given in appendix 1.

GENERAL PRINCIPLES FOR PARALLEL ALGORITHMS

Before discussing the quantum chemical algorithms it would be instructive to outline some general principles for programming distributed memory computers and to describe a few of the details involved in programming the Hypercube. When designing an algorithm for a single processor computer, the key to efficiency is to reduce the total number of operations. In contrast, for parallel algorithms the goal is to divide the problem into equal sized tasks that can run concurrently. Unfortunately these two goals are often conflicting so that the best serial algorithm for a given problem is not necessarily the best starting point for an efficient parallel algorithm.[36] This means that it will often not be adequate simply to modify existing programs. Instead, the problem should be carefully reconsidered with special attention to how the task can be subdivided.

When dividing a problem into subtasks a number of constraints must be kept in mind. Most importantly, each subtask must be nearly the same computational size. That is, the computational load on each processor should be evenly balanced. Another consideration is how much communication will occur between processors. Obviously some communication must occur if a useful task is to be carried out, but the time the processor spends communicating is wasted and will decrease the performance relative to a single processor algorithm. An additional consideration is that communication should be carried out in an orderly, organized manner. This is to avoid having one communication channel getting overloaded (thereby becoming a bottleneck) and to insure against the more serious problem of processor deadlock. Deadlock is the situation where a ring of processors becomes stuck because each processor in the ring is waiting for a message from the previous processor.

Given these constraints, what is the best strategy for dividing a problem into concurrent tasks? One alternative would be to write a unique program for each

individual processing element. However, this approach is unsatisfactory for a number of reasons. One problem is that such an implementation will only work on a fixed number of processors, so that the performance could not be improved by adding more processing elements to the computer. Moreover, it would be prohibitively difficult to write separate programs for each of the tens or hundreds of processors available in most distributed memory computers.

A better strategy for subdividing a problem is to have one "control program" (usually running on a separate "host" processor) coordinating the activity of a single "slave" program which runs on all of the remaining processors. This allows the use of a flexible number of processors since at run time the host program can read in the number of available processing elements and adjust the load on each processor accordingly. Also, this requires the writing of only two separate programs.

The first step in this strategy is to find one or more "dimensions" along which the problem can be divided. These dimensions are usually an adjustable parameter of the calculation, such as the number of particles in a molecular dynamics problem, the number of basis functions in a quantum chemistry calculation, or the order of the matrices in a linear algebra computation. The subdivision of the task onto the processors then occurs along these dimensions. For example, if the problem involves the construction of a large matrix where each matrix element is the result of the same operations on different data sets, then the problem naturally divides up with each processor computing a equal sized sub-block of the matrix (see figure 2).

Since most problems have a number of such dimensions careful thought should be given to which are subdivided. The choice of dimensions requires consideration of both constraints dictated by the problem, such as load balancing and communications costs, and constraints dictated by the computer, such as the number and connectivity of the processors. For example, consider the simulation of the motion of a

set of mutually interacting particles. The problem can be divided so that each processor is assigned either to propagate a fixed set of particles (regardless of where they are located) or to propagate all particles in a given region of space (regardless of how many particles it contains). The latter strategy is best for cases with a relatively uniform particle density -- so that the computational load is balanced, and for cases with short-ranged interparticle forces -- so that communication between processors is minimized. In order to avoid communication problems, however, this strategy requires a processors connectivity that supports a regular three dimensional mesh.

As can be seen from this example there is no simple recipe for devising an efficient parallel algorithm -- each problem requires careful consideration of constraints described above. In the following sections an effort has been made to clarify the steps in developing a efficient parallel algorithm by providing a step-by-step description of the development of these algorithms.

PROGRAMMING THE HYPERCUBE

At the present time programming the Hypercube is much like programming a serial computer. The programs for both the host and node processors are written in either FORTRAN or C (all programs described in this thesis were written in FORTRAN). The interprocessor communication is facilitated by a set of routines which handle all of the communication protocol. Most of the internode communication is carried out by with four routines: *send* and *recv* for asynchronous communication and *sendw* and *recvw* for synchronous communication. These routines take the same set of arguments: a channel number, a message type, the starting address and length of the message, and the destination node (or the sending node in the case of *recv* or *recvw*). The message type is a user specified integer which identifies the message. A message will be received by a processor only if its type matches that specified in

the *recv* or *recvw* call.

The synchronous communication routines (*sendw* and *recvw*) block the operation of the program until the message has been received in the case of *recvw* or copied out of the user space in the case of *sendw* . The asynchronous routine *recv* simply queues a request for the receipt of a message. Similarly the *send* routine simply initiates the transmission of a message before returning to the program. The host communicates to the node processors via two synchronous communication routines: *sendmsg* and *recvmsg* . Although the routines described here are specific to the hypercube, they are sufficiently simple and general that similar routines will probably be standard on future distributed memory computers.

The use of these routines is illustrated by the simple Hypercube program in figure 3. The host program reads in the number of processors to be used (*nproc*) and an array of numbers (x) equal in length to the number of processors. The host then sends out each of these numbers to a separate processor. The nodes receive the number, square it, and send it back to the host where the numbers from all the processors are summed together (*total*).

Finally, a few hardware features unique to the Hypercube should be considered since they will affect the design of the programs to be discussed. Most important of these is the limited memory available on the processing nodes. Although there is 512 Kbytes (thousand bytes) of memory available on each node, the node operating system requires nearly 200 Kbytes leaving only 320 kbytes of usable memory. Since the nodes have no access to mass storage and there is no provision for program overlay, all of the programs and data for a given calculation must fit into this limited space. This limitation will be discussed further when the benchmarks are presented.

A second item requiring special consideration is the Hypercube's very low ratio

of processing speed to communication speed (see appendix 1). In its standard configuration, the Hypercube can send a word (8 bytes) of data in nearly half the time it takes to carry out a floating point operation. This will clearly not be the case for future distributed memory computers and a vector processing board is now available for the Hypercube which boosts the processing speed by a factor of 100 (without increasing the communication speed). For these reasons the algorithms were designed assuming a relatively high processing speed to communication speed ratio.

Chapter II: Parallel Algorithms for Molecular Quantum Mechanics

INTEGRAL EVALUATION

An important consideration in developing a package of quantum chemistry programs is that most calculations involve running a series of these programs in sequence (for example, an integral evaluation followed by an SCF calculation). Hence, in order to optimize the efficiency of the overall computation it is necessary to design the set of programs to facilitate the most computationally demanding steps. This can be complicated because it often means choosing less than optimal algorithms for the simpler steps. The computational complexity of a number of common quantum chemical procedures are given in the following table (n = number of basis functions).

Computational complexities	
Procedure	Complexity
Integral evaluation	$O(n^4)$
SCF	$O(n^4)$
Integral transformation	$O(n^5)$
MPPT (second order)	$O(n^4)$
MPPT (third order)	$O(n^6)$
MPPT (forth order)	$O(n^7)$
CISD	$O(n^6)$

The first step in most quantum chemical calculations is the numerical evaluation of various integrals. There are three sets of one electron (two index) integrals: the nuclear repulsion, kinetic energy and overlap integrals. Each of these sets contains $\frac{n^*(n+1)}{2}$ integrals where n is the number of basis functions. The remaining integrals are the two electron (four index) integrals (equation I.4). This set is much larger than the one electron integral sets, containing $\frac{n^4}{8}$ symmetry distinct integrals. Since the calculation of the two electron integrals vastly dominates the integral evaluation step, the emphasis should be to parallelize this step. The computation of the one electron integrals will be carried out on the host processor.

In principle the parallelization of the two electron integrals is straight forward; each processor computes an equal sized subset of the total integral list. However, the distribution of the integrals onto the processor is complicated by considerations of how the integrals will be used in subsequent computational steps. In the development of this integral program two common sequences of quantum chemical computational procedures were considered. The first of these (the most common of all quantum chemical computations) is simply an SCF energy calculation. The second sequence is an SCF calculation followed by a post-SCF procedure requiring an integral transformation.

Integral distribution for SCF calculations

The computational complexity of the integral evaluation and SCF steps are the same, so neither step should be given dominant consideration. Further, since an efficient SCF algorithm can be designed which will work with arbitrary integral distributions, it is necessary only to focus on efficient load balancing of the integral evaluation. The simplest method to distribute

equally the integrals is to have each processor loop over all symmetry distinct integrals and select a unique subset to calculate and store. This can be accomplished by the segment of program given below:

```
ICOUNT = 0
JCOUNT = 0
DO μ = 1, NBASIS
 DO ν = 1, μ
  DO λ = 1, μ
   DO σ = 1, λ ( ν if μ= ν )
    ICOUNT = ICOUNT + 1
    IF ((MOD(ICOUNT,NUMPROC)).EQ.PROC_ID ) THEN
    EVALUATE INTEGRAL ( μν|λσ )
   END DO ( σ)
  END DO ( λ)
 END DO ( ν)
END DO ( μ)
```

Each processing node has a unique identification number (*PROC_ID*) ranging from 0 to the total number of processors minus 1. An integral is evaluated if the integral's cardinality (*ICOUNT*) modula the total number of processors (*NUMPROC*) is equal to the node's identification number.

This scheme guarantees equal balancing of the total number of integrals on each processor. A difficulty arises, however, since some basis functions represent more complex atomic orbitals than others (p or d orbitals as opposed to s orbitals) so that the computational cost for evaluating each integral is not the same. This means that even though each processor has an equal number of integrals, the net computational load may not be evenly balanced. Unfortunately a more exact load balancing would be very difficult to achieve. Even if the exact computational cost of each integral type was known in advance, equal distribution of the load would be an extremely difficult computational task. (This task is equivalent to the "multidimensional knapsack" problem which is in the class of NP complete problems widely

believed to have an exponential computational complexity).[37]

A possible alternative is to use dynamic load balancing. In this strategy a certain number of integrals are held back and then are dealt out to the first node processors to finish their initially assigned integrals. The decision was made not to implement dynamic load balancing in the integral evaluation step since it would greatly increase the complexity of the program and take up much needed memory space. Moreover, the simple scheme proposed evenly distributes both the computationally simple and complex integrals so that for large problem sizes the load balancing should be quite good. (This is confirmed by benchmark results given in the following chapter.)

Integral distribution for post-SCF calculations

If the two electron integrals are eventually to be transformed, a number of complicated constraints are placed on the distribution scheme. The transformation is simplified if the integrals are available in continuous segments (rather than scattered as in the method previously described). Moreover, the communication costs in the transformation step are greatly reduced if some integral redundancy is allowed.

For reasons that will be explained in the description of the transformation step, a good way to distribute the integrals is to map them onto a two dimensional rectangular array of processors (a subtopology of the hypercube connectivity). A simple way to carry out this mapping is to choose two of the integral's four indices as the coordinates of the processing node which is to evaluate the integral (figures 4-7). If there are more basis functions than processors along one of the dimensions then more than one basis function per processor is assigned along that dimension (see figures 6 and 7).

Due to the inherent symmetry of the two electron integrals, there are only two symmetry distinct pairs of indices which can be chosen: the last two indices λ,σ , and the second and fourth ν,σ. The choice of which two indices to use for the integral distribution is determined by what will be done with the transformed integrals. Figures 4-7 illustrate these two possibilities, both for the case where the number of basis functions matches the number of processors in each dimension (figures 4 and 5) and the case where the number of basis functions exceeds the number of processors along each dimension (figures 6 and 7).

These two distribution schemes involve the calculation of different numbers of integrals. The first scheme easily allows the use of one of the three index symmetries (permutation of the first two indices) so that there is a four fold redundancy in the integrals calculated. The second scheme does not allow the easy use of any of the integral symmetries so that all N^4 integrals are calculated. Of course the calculation of redundant integrals means that the integral evaluation step is non-optimal; however, this integral distribution will facilitate the more computationally complex integral transformation and post-SCF steps. Both of these distribution schemes will give good load balancing for sufficiently large problem sizes.

Two electron integral evaluation

Methods for the numerical evaluation of two electron integrals have been the focus of intense research efforts for the past three decades[38] and are still a subject of much interest.[39] The result of these efforts have been a large number of computationally efficient methods for integral evaluation. Although the integral evaluation scheme chosen for the Hypercube was dictated by memory considerations, the question of which methods are optimal

for parallelization should be addressed.

In the most efficient integral evaluation schemes two strategies are used to improve performance. One is the precomputation of large tables of repeatedly used numerical values. The other strategy is to compute the integrals in large groups to eliminate the redundant computation of partial terms. Both of these present special difficulties for parallel implementations which must be carefully considered.

A potential problem with the precomputation of lookup tables is that it does not use memory very efficiently since each of the processing nodes would have to store a copy of the table. Another concern is the manner in which the table is computed. It is obviously undesirable to have each processor calculate the entire table; however, if each processor computed unique segments, a large amount of communication would be required to fully distribute the table.

The most common approach to the second strategy is to calculate the integrals in complete shell blocks. A shell block is the set of all integrals over common atomic shells. For example, a complete $(\phi_p \phi_s \mid \phi_s \phi_s)$ block contains the following three integrals: $(\phi_{p_x} \phi_s \mid \phi_s \phi_s)$, $(\phi_{p_y} \phi_s \mid \phi_s \phi_s)$, and $(\phi_{p_z} \phi_s \mid \phi_s \phi_s)$. The following table lists the sizes of other shell blocks.

Shell block	Size
(s s | s s)	1
(p s | s s)	3
(p p | s s)	9
(p s | p s)	9
(p p | p s)	27
(p p | p p)	81

The rationale for this strategy is that the mathematical expressions for each integral in the shell blocks contain many common parts, so that if the entire block is calculated at once, many potentially redundant steps are avoided. This approach has proved to be very efficient on serial computers, but a number of problems arise when attempting to construct a parallel version of this scheme. These problems stem from the fact that to be efficient, the computation of the integrals in an individual shell block cannot be split across processors. This leads to the possibility of very poor load balancing due to the large discrepancies in the sizes of the shell blocks. A further difficulty that arises if the integrals are distributed by shell block is that the resulting distribution will complicate subsequent integral transformation and post-SCF steps. Despite the difficulties sited here, these two strategies potentially offer very large increases in the efficiency of the integral evaluation step, so despite their drawbacks, these methods should be carefully considered in future implementations of parallel integral evaluation programs.

As mentioned earlier, the prime concern in developing an integral program for the Hypercube was to conserve memory (since a fast integral evaluator would be of little use if there was no memory remaining to store the integrals.) Hence the decision was made to write an new integral evaluation program rather than use one of the available programs. The expressions used for both the one and two electron integrals are those derived by Taketa, Huzinaga, and O-ohata.[40] These expressions have the form of sums of products involving numerical constants, input values (angular momenta, internuclear distances), and the so-called associated F-function:

$$F_m(t) = \int_0^1 t^{2m} \exp(-xt^2)dt \qquad \text{(II.1)}$$

An advantage of this method over the more efficient techniques is that the expressions are completely general so that integrals can be calculated over basis functions of arbitrarily high angular momentum.

In the implementation on the Hypercube, each integral is evaluated individually, so that no efficiency is gained by avoiding redundant computation. The overall size of the two electron integral program based on this method is 55000 bytes. In comparison, a standard two electron integral package (excluding lookup tables) requires nearly 800,000 bytes. For the simplest two electron integrals, $(\phi_s \phi_s \mid \phi_s \phi_s)$, the Hypercube implementation is about a quarter the speed of the larger program.

SELF CONSISTENT FIELD CALCULATION

The SCF procedure is more complex and less homogeneous than the integral evaluation and hence a much more difficult task to parallelize. The first step is to look in detail at the various steps involved and the associated computationally complexity. The steps involved in an SCF iteration for a closed shell system are listed below.

1) Generate initial guess for the density matrix.

2) Form two electron part of the Fock matrix:

$$F_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma}[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\lambda|\sigma\nu)] \qquad \text{(II.2)}$$

3) Add kinetic and potential energy one electron integrals:

$$F_{\mu\nu} = F_{\mu\nu} + K_{\mu\nu} + V_{\mu\nu} \qquad \text{(II.3)}$$

4) Transform the Fock matrix using the overlap integrals:

$$\mathbf{F}^{\tau} = \mathbf{S}^{-1/2}\,\mathbf{F}\,\mathbf{S}^{-1/2} \qquad \text{(II.4)}$$

5) Diagonalize the transformed Fock matrix to get the SCF vector C and the orbital energies ε:

$$\mathbf{F}^{\tau}\,\mathbf{C}^{\tau} = \varepsilon\,\mathbf{C}^{\tau} \qquad \text{(II.5)}$$

6) Back transform the SCF vector:

$$\mathbf{C} = \mathbf{S}^{-1/2}\,\mathbf{C}^{\tau} \qquad \text{(II.6)}$$

7) Form new density matrix (NOCC = # occupied orbitals):

$$P_{\mu\nu} = 2 \sum_{i}^{NOCC} C_{\mu i}\, C_{\nu i} \qquad \text{(II.7)}$$

Steps 2-7 are repeated until the density matrix is converged. The computational complexity associated with each of these steps is given in the following table.

Computational complexities	
Step	Complexity
1	$O(n^2)$
2	$O(n^4)$
3	$O(n^2)$
4	$O(n^3)$
5	$O(n^3)$
6	$O(n^3)$
7	$O(n^2)$

It is obvious from the data in this table that for large problem sizes the SCF computation will be dominated by step 2, the formation of the two electron portion of the Fock matrix. This result is corroborated by actual timings on serial SCF programs.

Hence, the logical starting point in the parallelization of the SCF procedure is to distribute the computation of the two electron Fock matrix. The two components required to form this term are the density matrix and the two electron integrals. Since the two electron integrals have already been distributed onto the nodes by the integral program and these integrals do not change with each iteration, a sensible procedure for forming the Fock matrix is to distribute the density matrix to the nodes each iteration and then calculate partial Fock matrices from the resident two electron integrals. These partial Fock matrices are then summed together on the host processor where the new density matrix is formed and broadcast back to the nodes. A flow diagram of this procedure is given in figure 8.

The method used to form the partial two electron Fock matrices is dependent on the integral distribution scheme used in the integral evaluation step. If the integrals were distributed for just an SCF calculation the formation of the partial Fock matrices is straight forward since a nonredundant list of two electron integrals is evenly distributed onto the processors. This procedure on each node involves simply looping over all of the resident integrals

calculating their contributions to the Fock matrix. Since only the symmetry distinct integrals have been stored, each integral will contribute to between one and eight Fock matrix elements. Figure 9 lists all Fock matrix element contributions from symmetry distinct integral types.

If an integral transformation is to be carried out, the formation of the partial Fock matrices is more difficult because of the more complex integral distribution. The difficulty is that redundant integrals are stored so that special care must be taken that redundant contributions are not made to the partial Fock matrices. This consideration greatly complicates load balancing. Since the integral distribution is determined by two of the integral indices, most schemes for selecting symmetry distinct integrals will preferentially load certain processors. However, a scheme which works well for most cases is straight forward. Consider the simplest integral loading (as in figure 4) where a processor with coordinates $\underline{k},\underline{l}$ holds the integral block $(ij|\underline{k}\underline{l})$ where $i,j = 1$ *to* n. If $\underline{k} < \underline{l}$ the node processes all symmetry distinct integrals for which $i+j$ is odd. If $\underline{k} > \underline{l}$ then the node processes all symmetry distinct integrals for which $i+j$ is even. Of course if $\underline{k} = \underline{l}$ then all of the symmetry distinct integrals must be processed. Hence this strategy will overload the diagonal processors ($\underline{k} = \underline{l}$), but since for most actual cases each node will have many integral blocks (as in figures 6 and 7) this load imbalance should not be serious.

As indicated in the flow diagram (figure 8), the SCF procedure requires the communication of all of the partial Fock matrices from the nodes to the host and the subsequent rebroadcast of the new density matrices. If all of the communication were to be carried out directly between the nodes and the host, the host's communication channels would be overloaded and become a

bottleneck. What is needed is to carry out as much of the communication in parallel as is possible and to avoid having an individual processor sending or receiving more than one message at a time. Fortunately there is a simple scheme utilizing a so-called broadcast tree which fulfills these conditions. The broadcast of a message from the host to all of the nodes in a three dimensional hypercube is shown in figure 10.

Basically, the scheme involves the sequential broadcast of the message from all of the nodes in a N dimensional hypercube to all the nodes in an $N+1$ dimensional hypercube. Hence, it takes a total of six communication steps to broadcast a message to all of the nodes in a five dimensional (32 processor) Hypercube. (This includes the initial communication of the message from the host to node 0.) Similarly, if a result is to summed from the nodes onto the host, the reverse procedure can be carried out: each node in an N dimensional cube receives a message, sums it into his own result, and broadcasts this message to the corresponding processor in an $N-1$ dimensional cube.

Although parallelizing the most computationally complex part of the SCF procedure is clearly the first step, there are a number of reasons also to distribute the less computationally complex tasks. One obvious reason is that for sufficiently large problem sizes, even the $O(n^3)$ steps in the SCF procedure will take a prohibitively long time on the host processor. A second reason is that for problems running on computers where the number of processors (m) is comparable to the number of basis functions (n), the complexity of the Fock matrix formation $O(n^4/m)$ reduces to $O(n^3)$ so that the other $O(n^3)$ steps will dominate the computation time. (This is the case for some of the benchmark calculations on the Hypercube where the memory size severely

limits the number of basis functions.)

There are two types of $O(n^3)$ steps occurring in the SCF procedure: the matrix multiplications and the diagonalization of the Fock matrix. The matrix multiplies are fairly easy to distribute, but the diagonalization is much more difficult to parallelize efficiently. Although a parallel diagonalization has not yet been implemented, the method for doing this is discussed here.

PARALLEL MATRIX DIAGONALIZATION

Due to their tremendous importance in a vast number of mathematical and scientific applications, numerical methods for the computation of eigenvalues and eigenvectors of symmetric matrices have been the subject of intense study for more than century. A large number of very efficient serial algorithms have been developed which are able to compute the eigenvectors of an $n \times n$ matrix in $O(n^3)$ time[41] . Not surprisingly, in recent years a lot of effort has been put into the study of parallel matrix diagonalization algorithms. One conclusion of this work is that the very best serial algorithms (e.g. Householder tridiagonalization followed by QR iterations) are not as well suited for parallelization as the older, simpler methods.

One very old algorithm which seems especially promising for parallel applications is the Jacobi method (first described in 1846).[42] In this scheme a symmetric matrix **A** is diagonalized by a series of plane rotations. Each rotation is carried out such that it annihilates $n\frac{(n-1)}{2}$ of the off diagonal matrix elements. Since the zeroed off diagonal matrix elements will not necessarily remain zero for successive rotations, the method is iterative. If m such rotations are necessary to zero all of the off diagonal elements of **A**, then:

$$\mathbf{U}\,\mathbf{A}\,\mathbf{U}^{+} = \mathbf{R}_m\,\mathbf{R}_{m-1}\,\cdots\,\mathbf{R}_1\,\mathbf{A}\,\mathbf{R}_1^{+}\,\mathbf{R}_2^{+}\,\cdots\,\mathbf{R}_m^{+} \qquad \text{(II.8)}$$

where **U** is the matrix of eigenvectors of **A** and $\mathbf{R}_m$ is the m^{th} successive plane rotation matrix. The eigenvalues are given by:

$$\mathbf{U} = \mathbf{R}_m\,\mathbf{R}_{m-1}\,\cdots\,\mathbf{R}_1 \qquad \text{(II.9)}$$

and the rotation matrix elements at each iteration are given by:

$$R_{ii} = R_{jj} = \cos(\alpha_{ij}) \qquad \text{(II.10)}$$

$$R_{ij} = -R_{ji} = \sin(\alpha_{ij}) \tag{II.11}$$

$$\textit{where:} \quad \tan(2\alpha_{ij}) = 2\frac{A_{ij}}{A_{ii}-A_{jj}} \tag{II.12}$$

Since the computation of the eigenvectors simply corresponds to a series of matrix multiplications, there are a number of possible strategies for parallelization of the Jacobi algorithm. One approach has been investigated by Sameh [43] and Whiteside, et al.[44] This scheme is based on the observation that there exist sets of plane rotations that each annihilate different matrix elements without changing the values of the matrix elements annihilated by other rotations in the same set. Since all rotation matrices in each set commute with each other, the rotations can be carried out independently. For an *n by n* matrix there are $\frac{n}{2}$ rotation matrices in each of these sets so that $\frac{n}{2}$ off diagonal elements could be eliminated simultaneously. Thus, this algorithm could efficiently utilize $\frac{n}{2}$ processors in parallel. For the SCF procedure described here, $\sim n^2$ processors are used to form a Fock matrix of order n. However, this diagonalization routine would only use $\sim n$ processors. Hence, this algorithm is poorly balanced for the SCF procedure.

Fortunately another parallel Jacobi diagonalization algorithm has recently been developed which allows the efficient use of much larger numbers of processors.[45] In this scheme the matrix is divided into 2 x 2 submatrices and mapped onto a two dimensional square array of processors, so that an *n by n* matrix would map onto an $\frac{n}{2}$ *by* $\frac{n}{2}$ array of processors (see figure 2). During each iteration the diagonal processors (i.e. those containing diagonal matrix elements) calculate the rotation matrix necessary to diagonalize their resident 2 x 2 matrix. This rotation matrix is broadcast to the other

processors which carry out the necessary rotations on the off diagonal terms. After one such complete rotation has been carried out, adjacent processors exchange rows and columns so that the diagonal processors receive new off diagonal matrix elements of the rotation is again carried out. The method has been empirically found to require $O(\log(n))$ such rotations with each rotation requiring $O(n)$ computing time, yielding a net computational complexity of $O(n\log(n))$. Although this method has not yet been implemented on the Hypercube, its low computation cost, orderly interprocessor communication, and efficient use of ~ n^2 processors make it a good candidate for the Fock matrix diagonalization.

TWO ELECTRON INTEGRAL TRANSFORMATION

The two electron integral transformation is the most computationally complex of the steps so far considered. Hence, much of the concern in the design of the algorithms for the previous two steps was to facilitate the efficiency of this computation. This procedure involves the transformation of all four indices of the two electron integrals. The most compact expression for this step is:

$$(ij\,|kl) = \sum_{\mu}\sum_{\nu}\sum_{\lambda}\sum_{\sigma} C_{\mu i} C_{\nu j} C_{\lambda k} C_{\sigma l} (\mu\nu\,|\lambda\sigma)$$

where the Roman letters represent MO indices and the Greek letters AO indices. Since this $O(n^4)$ must be carried out for all of the $\frac{n^4}{8}$ integrals, the total complexity as written above is $O(n^8)$. However, this transformation can be rewritten as a series of one index transformations (or "quarter transformations"). For example, the transformation of the fourth index, σ can be carried out as an autonomous step:

$$(\mu\nu\,|\lambda l) = \sum_{\sigma} C_{\sigma l} (\mu\nu\,|\lambda\sigma)$$

Each of these quarter transformations involve n^5 multiplications, so that the entire sequence of one index transformations requires $4n^5$ multiplications. If the 8 fold index symmetry is exploited, this total can be reduced to $\frac{5}{2}n^5$. More sophisticated transformations have been developed which involve the decomposition of each quarter transformation and require a total of only $\frac{25}{24}n^5$ multiplications.[46] However, as pointed out previously, the optimal serial algorithms are usually too complex to be converted into an efficient

parallel implementation. Instead, it is usually better to begin with a simple expression for the problem and to look for dimensions along which to divide the task.

In the transformation step a natural choice for these dimensions would be one or more of the two electron integral indices. The advantage of this choice is that all of the integrals needed to complete a given quarter transformation would be present along a single processor or along a single dimension of the processor array. To illustrate this concept consider the mapping of the two electron AO integrals onto a one dimensional array of processors (i.e. a loop). (A three basis function example is shown in figure 11.) The integral blocks are assigned to processors on the basis of their fourth index, σ. Each integral block contains integrals for all symmetry distinct combinations of μ, ν, and λ: $\mu, \nu = 1$ *to* n *with* $\mu \geq \nu$, and $\lambda = 1$ *to* n. Note that only one of the three axes of integral symmetry is used, so that a four fold redundancy of integrals is stored.

For this integral distribution, the first step is the transformation of the indices local to each processor, μ, ν, and λ (step 2 in figure 11). For example, on the first processor this involves:

$$(ij\,|k\,1) = \sum_{\mu}\sum_{\nu}\sum_{\lambda} C_{\mu i} C_{\nu j} C_{\lambda k} (\mu\nu\,|\lambda\,1)$$

(Of course in an actual implementation this would be carried out as three quarter transformations.) Due to the integral distribution this step can be carried out without any interprocessor communication.

All that remains is the transformation over the final index, σ. Since each processor holds integral blocks with only a single value of σ and this final quarter transformation requires a summation over this index, this step

will require interprocessor communication. Because of the distribution scheme, this can be carried out in a series of orderly communication steps as follows (step 3 in figure 11). First, each processor sends a copy of the partially transformed integrals to its neighboring processor. Each processor then sums a contribution from the visiting integral block into a buffer:

$$(ij\,|k l_{res}) = (ij\,|k l_{res}) + C^{l_{res}}_{\sigma_{vis}}\ (ij\,|k\,\sigma_{vis})$$

where the subscripts *res* and *vis* indicate indices associated with the resident and visiting integral blocks. Note the assumption that the processors will retain the same AO and MO integral blocks: $l_{res} = \sigma_{res}$.

This step is repeated until the integral blocks have been passed completely around the loop and have arrived at their initial nodes. At this point the transformation is complete since each processor has received all the necessary contributions for the last quarter transform.

Obviously this one dimensional mapping could be extended to two, three, or four dimensions by simply distributing more of the integral indices in the way σ was distributed in the one dimensional case. The choice of how many dimensions to use is essentially that of the granularity of the parallelism.

The granularity of the parallelism refers to the amount of independent computation that each processor performs before it must communicate with another processor. If a large amount of processing is performed, for example a pass through some outer loop of the program, then the granularity is "course." If only a small amount of processing is performed between communication steps, for instance only a single arithmetic operation, then the parallelism is "fine grained." Because of the relatively slow interprocessor

communication channels on the Hypercube (probably this will be a problem on all distributed memory machines) fairly course grained communication is favored. Otherwise, the communication overhead will be much more time consuming than the useful computations performed.

Other considerations in the choice of dimensionality are the number of available processing nodes and the total amount of memory on each node. If the integrals are mapped onto a one dimensional grid (see figure 11), the largest number of processors that can be utilized is equal to the number of basis functions, n, and each node must store ~ n^3 integrals. Similarly, for a two dimensional mapping, the maximum number of usable nodes would be n^2, with each node storing ~ n^2 integrals. The following table lists the relevant data for each of the possible mappings.

Integral Distributions			
Dimensions	# usable nodes	Integrals per node	Operations per communication
1	n	n^3	n^3
2	n^2	n^2	n^2
3	n^3	n	n
4	n^4	1	1

For most chemically interesting problems, the number of basis functions is not more than a few hundred. In comparison, the Intel Hypercube is available with up to 128 processors and other computers are available with thousands of processors. Hence, the one dimensional transformation scheme would not be able to utilize fully the available hardware. However, the two dimensional mapping would be able to use efficiently the large parallel

computers available in the foreseeable future while still maintaining a fairly course grained parallelism.

A final issue is the topology of interprocessor interconnection assumed by the parallel implementation. Although these programs are being implemented on a machine with hypercube connectivity, it is not necessarily prudent to assume the presence of that particular interconnection in this algorithm. If an efficient algorithm can be developed that requires only very simple topologies, such as one or two dimensional meshes, the algorithms will run on a wider range of parallel architectures.

On the basis of the considerations listed above, the two dimensional transformation scheme was chosen for implementation. This choice clarifies the reasons for the post-SCF integral distributions described in the integral evaluation section (figures 4-7). The two indices which indicate the processor coordinates are those which require interprocessor communication to transform, (analogous to σ in the one dimensional case). The transformation procedure for the two post-SCF integral distribution schemes, $\lambda\sigma$ (figures 4,6) and $\nu\sigma$ (figures 5,7), are essentially identical, differing only in the order in which the indices are transformed. Hence, in the algorithm described here, only the $\lambda\sigma$ distribution will be considered. Further, in the following description the processing grid is assumed to be a square torus of processors (m by m) where the number of basis functions, n, equals the number of nodes along each dimension (i.e. $m = n$). These constraints are not inherent in the algorithm which will work efficiently for any number of basis functions provided ($n \geq m$) and on any torus supported by the interprocessor connectivity. (In general a hypercube architecture has as a subtopology a 2^i x 2^j rectangular torus where $i+j \leq$ the total hypercube dimen-

sionality.) The extension to cases with rectangular processing grids and cases where $n > m$ is straight forward.

At the outset of the integral transformation the integrals are distributed onto the processing nodes on the basis of their final two indices, $\lambda\sigma$, as shown in figure 4. The transformation matrix **C** is the final converged SCF vector which is broadcast from the host to all of the nodes using a broadcast tree.

The first step is the transformation of the first two indices, μ and ν (figure 12). This step is carried out as two quarter transformations. If each processor is assumed to hold only one $\lambda\sigma$ block and no integral symmetry is considered, then each quarter transformation requires n^3 multiplications. This yields a total of $2n^3$ for this entire step. However, if the $\mu\nu$ symmetry is exploited then the number of multiplications is reduced to $\frac{3}{2}n^3$.

The final two quarter transformations are more complicated since they will require interprocessor communication. Analogous to the transformation over σ in the one dimensional case, these last two steps will require communication first across the processor rows (figure 13) and then down the processor columns (figure 14). Although these two steps are in principle straight forward, a number of details must be carefully considered in order to use efficiently memory and to avoid communication bottlenecks.

There are two general methods for carrying out the communication steps. The processors could either pass around MO integrals (with initial values of zero) that accumulate contributions from each of the other processors as they circumnavigate the row, or they could pass around raw (half transformed) AO integrals and accumulate MO integrals locally from each of the AO integral subsets as they pass by. (The latter strategy was described

for the one dimensional example).

The chief distinction between these two strategies is the type of interprocessor communication allowed by each. In the first scheme all communication must be synchronous. That is, communication is not carried out while the nodes are involved in other computation. The reason for this is that each of the MO integral buffers must remain on each node until all of contributions from the resident integral set has been added in. Only when all of these computations are completed is the integral buffer ready to be passed on to the next processor.

In contrast, when the raw AO integral sets are passed, the processor needs only to make a copy of the integral set before passing the integrals on to the next processor. This second scheme is obviously more efficient since it allows the overlap of communication and computation. However, the asynchronous scheme has some potential pitfalls. Since the integral buffers are passed on by each processor immediately, slow processors could potentially queue up many unreceived messages which can cause erratic behavior on the Hypercube. Nevertheless, in order to study the performance enhancement gained by asynchronous communication the second scheme was implemented in the final version of the integral transformation.

The code given in figure 15 summarizes the operations performed during the third quarter transformation. The routine is passed the resident block of half transformed integrals (*xints*), the SCF vector (C), the number of basis functions (*nbasis*), and the value of the third integral index for the resident AO (*mylambda*) and MO (k) integral blocks. Note that for the example in figures 12-14, *mylambda* = k .

In order to allow asynchronous communication, the scratch array *buf* is

divided into two segments, each able to hold a full integral block. Two variables, *outpoint* and *inpoint*, hold pointers to the two halfs of *buf*. The half indicated by *outpoint* holds the integral block about to be processed, and the half indicated by *inpoint* receives the incoming integral block.

The subroutine begins by initializing the pointers and copying its integrals from the array *xints* into the half of *buf* pointed to by *outpoint* , zeroing *xints* as it does. The MO integrals will accumulate in *xints*. The routine then enters the main loop over the number of processors in each row, which equals *nbasis* for this example. Then first step is to send off a copy of the integral block that is about to be processed (stored in *buf* (*outpoint*)). Next, *recv* is called to initiate the receipt of a new block of integrals which will be stored in *buf* (*inpoint*) . After the communication steps are initiated, the contributions from the integrals in *buf* (*outpoint*) are summed into *xints*. Next the variable *lambda* is decremented in anticipation of the new integral block. Since it is possible that this communication is not yet complete, the routine calls *waitchan* which blocks execution until the integral block is completely received. At this point the cycle is ready to begin again except that the roles of the two halfs of *buf* are switched -- the newly received block of integrals will be sent on and processed, and the previously processed block will be overwritten by the incoming integral block. This switch can be accomplished by simply swapping the values of the pointers *inpoint* and *outpoint* .

After *iloop* has looped over all of the processors in the row, each processor has received all of the necessary contributions for the transformation of the third index. The only step remaining is the transformation of the final index σ . This step can be carried out in a way exactly analogous to the

transformation of the third index, except that the interprocessor communication is down the processor columns rather than across the processor rows.

For clarity of presentation several important details have been neglected in this description. For instance, the code as presented requires each half of the buffer space *buf* to be able to hold a full integral block. In fact, the transformation can be carried out with a smaller buffer, it simply requires more passes through a code similar to this, each pass completely transforming only a subset of the integral block.

Each of the last two quarter transformations requires $\frac{n^3}{2}$ operations on each of the processing nodes. Since only n sequential communication steps are required for each quarter transformation, communication time will not dominate the execution time except for very small problem sizes. Each of the n^2 processors carry out $3\frac{n^3}{2}$ operations for the first half transformation over μ and ν and n^3 operations for the final half transformation over λ and σ, yielding a total computational complexity of $5\frac{n^3}{2}$ on each of the nodes for the entire four index transformation.

As mentioned earlier, for problems where the number of basis functions is not evenly divided by the number of nodes along each dimension of the processor mesh, the number of integrals blocks assigned to each processor will be unequal and, hence, the computational load will be poorly balanced. An extreme example is shown in figure 6 where four nodes must process four integral blocks each and one node is assigned only one integral block. Since the lightly loaded processors will eventually have to wait for the more heavily loaded processors (during the third and fourth quarter transformations) the transformation of the $N = 5$ case will require the same amount of

time as the $n = 6$ case. Of course for larger problem sizes where the number of basis functions is much larger than the number of nodes along each dimension of the grid, this load imbalance will contribute a proportionally very small contribution to the total running time. Nevertheless, it is worth investigating methods to improve the load balancing.

MOLLER PLESSET PERTURBATION THEORY

The integrals produced by the transformation step have little intrinsic scientific value. They are useful only as input to subsequent computations. There are a large number of post-SCF procedures requiring the transformed integral list. The two most commonly used methods are configuration interaction and Moller Plesset perturbation theory. For reasons that will be discussed shortly the perturbation theory calculations are easier to parallelize so these were chosen as the first post-SCF methods to implement on the Hypercube. Considerations for the parallel configuration interaction algorithm will be discussed in a later section.

As described in the first chapter, Moller Plesset perturbation theory provides a series of additive corrections to the total electronic energy calculated by the SCF procedure. For all orders of the perturbation theory, these energy corrections have the same form: linear combinations of products of super-integrals formed from the transformed two electron integrals and coefficients formed from the SCF orbital energies (the eigenvalues of the converged Fock matrix). In principle this step is easy to parallelize. Each processor asynchronously forms contributions to the energy correction from its resident list of transformed integrals. When completed, these partial contributions are sent back to the host processor where they are summed together. The various details involved in this procedure will be covered in the following description of the second order Moller Plesset energy correction ($E^{(2)}$) . Extension to higher orders of perturbation theory is straight forward.

The standard expression for $E^{(2)}$ is:

$$E^{(2)} = \sum_{ab}^{occ} \sum_{rs}^{vir} \frac{|< ab \,||\, rs >|^2}{D_{abrs}}$$

This expression is meant to be used as a correction to unrestricted Hartree Fock (UHF)[24] energies and hence, the summations are over spin orbitals. However, the SCF program implemented on the Hypercube carries out restricted Hartree Fock (RHF) calculations so that spin independent spatial orbitals are produced. (The relative merits of UHF and RHF have been extensively debated.[47]) The conversion of the Moller Plesset energy expressions from spin to spatial orbitals is quite easy. The resulting $E^{(2)}$ expression for closed shell systems is:

$$E^{(2)} = \sum_{ab}^{occ} \sum_{rs}^{vir} \frac{(ar\,|bs)[2(ar\,|bs)-(as\,|br)]}{D_{abrs}}$$

where the integrals are the standard transformed two electron integrals and D_{abrs} is formed from the final SCF orbital energies:

$$D_{abrs} = \varepsilon_a + \varepsilon_b - \varepsilon_r - \varepsilon_s$$

Hence, in order to form contributions to $E^{(2)}$ the nodes need access to the transformed integrals and the orbital energies. Since the number of orbital energies is the same as the number of basis functions ($<$ 150), it makes sense to send a separate copy to each of the nodes.

The transformed integrals are already distributed onto the nodes by the transformation program; however, the formation of each term in $E^{(2)}$ requires the presence of on a single processor of a pair of integrals (corresponding to the pair of integrals required to form a single super-integral). These two integrals are related by the permutation of their second and fourth indices. An important question is whether these pairs of integrals will be available on the same processor or will interprocessor communication be necessary to collect them. Of course the answer depends on the final

distribution of the transformed integrals.

As described in the previous sections two possible post-SCF integral distribution schemes have been investigated. The first distributes the integrals on the basis of their third and fourth indices (figures 4 and 6), and the other distributes them on the basis of their second and fourth indices (figures 5 and 7). For the computational steps through the integral transformation the chief difference between these two is that the latter requires an additional factor of two redundancy in the computation, storage, and transformation of the integrals. However, for the computation of $E^{(2)}$ these two distributions will lead to very different algorithms.

For the first of these schemes, the appropriate pairs of integrals will not be together on the same processor. For example, in figure 4 the transformed integral (33|11) is assigned to the node in the upper left corner, while its partner (31|13) is assigned to the node in the upper right corner. This means that if the calculation of $E^{(2)}$ is to be carried out, some interprocessor communication will have to occur. The remaining question is whether this communication can be carried out in an orderly, efficient way.

In this integral distribution scheme a given node has MO integrals with all possible values for the i and j indices and with values for k and l indices determined by the processor's position in the grid. Since the integral pairs are related by the permutation of the second and fourth indices, an integral block on a particular processor will have to access all blocks with the same k index, but having all possible values for the l index. Due to the fact that the l index is used to assign the integral blocks to processor rows, the necessary integral blocks can be found on the other processors in the same column. Hence, the communication is very similar to that required in the final quarter

transformation; buffers must be passed around the columns of the processor grid.

To clarify this somewhat complicated explanation, consider again the 3x3 example used in the transformation section. After the transformation, the integrals are distributed onto the processors as shown in figure 4 The node in the upper right holds the integral block $(ij|11)$. In order to form the corresponding block of super-integrals, this block will have to be combined with all integrals of the form $(i1|1j)$ with i,j = 1,3. All of these integrals can be found in the first column of processors. A remaining question is how to assign $E^{(2)}$ contributions to the processors or, more generally, how to distribute the super-integrals. Since the super-integrals are defined:

$$< ab||cd> = (ac|bd)-(ad|bc)$$

a natural choice is to let the processor holding the first integral form and process the corresponding super-integral.

Note that for the calculation of the second and third order Moller Plesset energy corrections not all the super-integrals need to be formed. For example, $E^{(2)}$ only has contributions from super-integrals of the form $< ab||rs>$ with a,b occupied orbitals and r,s virtual orbitals. These constraints will limit the number of integral blocks that need be sent around the processors.

The "driver" subroutine for the calculation of $E^{(2)}$ is given in figure 16. Each processor loops over its resident integral blocks sending valid blocks down the processor column. For the calculation of $E^{(2)}$ valid blocks are those with k an occupied orbital and l a virtual orbital. (If a processor has no more valid blocks to send, it must still receive the remaining blocks being broadcast by the other processors in order to finish the contributions to its resident super-integrals.) After sending an integral block down the grid, each

node receives an integral block and immediately sends on a copy of this block by calling the subroutine *sendblkon*. Next, *contrb* is called which generates all super-integrals that can be formed from the resident and visiting integral blocks. Once formed, the super-integrals are not stored, instead they are immediately combined with the appropriate D_{ijkl} elements (which are formed as needed) and summed into the energy term. When all the nodes are finished calculating the partial $E^{(2)}$ terms, *hostsum* is called to sum the terms down a broadcast tree onto the host.

As mentioned earlier, the alternative integral distribution leads to a very different algorithm. In this scheme the integrals are assigned to processors on the basis of their second and fourth indices. In the 3x3 example (figure 5) the upper left processor holds the integral block $(i\,1\,|k\,3)$ which is by symmetry equivalent to $(i\,3\,|k\,1)$. Hence, with this integral distribution, no communication will be necessary to form the super-integrals. The algorithm for calculating $E^{(2)}$ is very simple. The routine simply loops over valid super-integrals, summing energy contributions into a buffer. When the loop is complete, the partial terms are summed onto the host processor.

Each of the two integral distributions has merits and disadvantages. The first allows efficient integral evaluation and transformation, but requires n sequential communication steps to form the super-integrals. In contrast, the second scheme allows the formation of the super-integrals without communication, but adds a factor of two to the number of integrals which must be calculated and transformed. Which of these two scheme is ultimately the best is dependent on the hardware being used. For some of the presently available computers, on which communication and processing speeds are comparable and on which memory is at a premium, the former scheme is best. How-

ever, for future parallel computers on which processing speeds will be much faster than communication speeds and memory will not be a limitation, the latter scheme will be optimal. Benchmarks results for both schemes are given in the next chapter.

CONFIGURATION INTERACTION

Although Moller Plesset perturbation theory (MPPT) and configuration interaction (CI) are both methods to correct for SCF's exclusion of instantaneous electron-electron correlation in the electronic wavefunction, the actual computational procedures involved are very different. As discussed in the previous section, the computation of MPPT energies involves summations over the transformed integrals, so the parallelization is fairly straight forward. In contrast, CI calculations involve the formation and diagonalization of the CI matrix $< \Phi |H| \Phi >$. This procedure is computationally difficult for both serial and parallel implementations since this matrix is very large -- of order at least 100,000 for most systems of chemical interest. Fortunately this matrix is sparse and only the first few roots are ever desired, so that efficient iterative diagonalization methods can be used.

A commonly used iterative diagonalization method is that of Davidson.[48] In this algorithm the true eigenvectors C_j are approximated by a set of k basis vectors b_i:

$$C_j = \sum_i^k \alpha_{ij} b_i$$

where k is much less than the order of the matrix. Each iteration involves the construction and diagonalization of a $k \times k$ matrix P to form a new set of coefficients α_{ij} . Additionally with each iteration a new basis vector b is added to the expansion. The construction of the matrix P requires the multiplication of each of the basis vectors by the matrix to be diagonalized. Since a new basis vector is added with each iteration, this large matrix multiply must be carried out each iteration.

While the Davidson procedure provides an efficient way to compute the first few roots of the CI matrix, as described above it does not alleviate the need to store the entire CI matrix. This can be avoided, however, by using a method known as direct CI. In direct CI as each matrix element contribution (a product of a transformed integral and a coefficient) is formed it is immediately combined with the appropriate elements in the guess vector. In this way when all of the CI matrix elements have been constructed the matrix multiply is complete.

When the transformed integral list is evenly distributed onto the node processors the direct CI method is well suited for parallelism. Each processor calculates the coefficients for its set of transformed integrals. As each of these partial matrix elements are produced, they are combined with the appropriate elements in the guess vector. When the nodes have processed all of their two electron integrals, the product vectors are summed together on the host. The new guess vector is then generated and broadcast to the nodes.

Although the algorithm should be easy to implement it has the drawback that each of the processors must hold a copy of the guess vector (which is the same length as the CI expansion) as well as its set of MO integrals. Hence, due to the limited memory currently available on the Hypercube the CI program has not yet been implemented.

Chapter III: Benchmark Results

INTRODUCTION

No matter how impressive the theoretical efficiency of an algorithm, the true test is its actual performance when implemented on a real computer. This is particularly true for parallel algorithms where unexpected bottlenecks can drastically reduce the anticipated performance. In order to test the algorithms presented in the previous chapter, they were implemented on a 32 processor Hypercube located at Sandia National Laboratories. Before presented the actual results, it would be worthwhile to describe briefly the theory of measuring the efficiency of parallel algorithms.

The principle measure of performance is the elapsed time necessary to solve a real problem of interest. However, in evaluating parallel programs the speedup gained by adding additional processors is often a more informative measure. The speedup of a parallel program run on k processors, S_k is defined by

$$S_k = \frac{t_{ref}}{t_k}$$

where t_k is the time necessary to complete the computation on a k processor parallel computer, and t_{ref} is some reference time for the computation. It is common to take t_{ref} to be the time necessary to perform the computation on a single processor of the type used in the multiprocessor. Therefore, the ideal case is $S_k = k$ indicating that the computation runs a factor of k times faster when running on k processors.

The first set of benchmark results are for the integral and SCF programs. Results are given for both standard and direct SCF calculations. Next, the results for complete integral, SCF, transformation, and second order Moller Plesset energy calculations are given. Finally, the results are presented of an extensive battery of benchmark calculations for the most computationally complex of the programs implemented, the two electron integral transformation.

As mentioned earlier, the chief constraint on the problem sizes which can be handled by the Hypercube is the relatively limited memory on each processing node. In the standard configuration, each node has 512 Kbytes of RAM. After the node operating system is loaded, 322 Kbytes remain for the user programs and data. The node programs for the integral and SCF calculations require 81 Kbytes leaving 240 Kbytes. Since an n basis function problem requires the storage of $\sim \frac{n^4}{8}$ integrals (each requiring 8 bytes of memory), the maximum size SCF calculation that can be carried out using all 32 nodes is:

$$\sqrt[4]{32 \times 240000} \approx 50 \textit{ basis functions}$$

(Of course for a direct SCF calculation there is essentially no limit on the problem size since no integrals are stored.) The full set of node programs (integral evaluation through Moller Plesset energy) require 119 Kbytes leaving 203 Kbytes for integral storage. The two integral distribution schemes investigated require the storage of $\frac{n^4}{2}$ and n^4 integrals, respectively, corresponding to maximum problem sizes of 35 and 30 on a full 32 node Hypercube.

This limitation can be alleviated by either adding more memory to each node or by adding more processing nodes (so that each has a smaller fraction of the total integral list). Both of these options are now available for the Hypercube. The total number of processing nodes can be increased to 128, and the memory per node can be upgraded to 4 Mbytes. All calculations carried out for this thesis were on the standard configuration Hypercube.

In order to determine the program speedups it is necessary to run the benchmark calculations on various sized subsets of the Hypercube processors. The Hypercube allows the user to specify easily how many nodes to use for a given calculation. A natural choice of subsets used for most of these benchmarks are the "sub-cubes" of lower dimensions having 2^n processors for $n=0$ *to* 5 . However, using fewer than the full 32 nodes decreases the total available memory, so that the size of the test cases will be dictated by the smallest processor array in the benchmark sequence.

In order to test the performance on both large and small cases two test problems were chosen. The first is 24 basis function C_2H_2 for which a full Moller Plesset calculation can be carried out on as few as 8 processing nodes. The second test case is 13 basis function H_2O which is sufficiently small to fit on a single processing node. The details of these test cases are given in appendix two.

SCF RESULTS

Since the host-to-node communication rate is about $\frac{1}{3}$ that for node-to-node, for calculations involving no post-SCF steps it is more efficient to have one of the node processors take the role of the host, receiving and diagonalizing the Fock matrix. (Note that this would be complicated for post-SCF

computations since the transformation requires a full rectangular grid.) This strategy has been employed for the SCF benchmarks, and hence the processor grids have size $2^n - 1$ (n = 1 to 5).

The integral and SCF benchmark results for the C_2H_2 test case are given in the tables in figure 17. Each of these tables has three columns of data. The first contains results for the two electron integral evaluation. The second gives results for the SCF step and the third lists results for the construction of the partial Fock matrix.

The first table gives the total timings in seconds. Obviously the timings in the second and third columns vary from node to node (unless load balancing is perfect). The timings given are those for the slowest processors. The second table lists the speedup ratios derived from the timings in the first table. The values should be compared with the numbers in the "ideal speedup" column. These data show that the integral evaluation and Fock matrix formation steps have nearly perfect speedups. The slightly superlinear speedups in the table are due to changing load balance for different sized test cases.

Although the results for the Fock matrix formation step are very good, the speedup for the full SCF calculation is not very encouraging. The efficiency on 31 processors is less than 30%. Since the parallelized portions show good speedup, the problem must be that the serial portion (the communication and diagonalization of the Fock matrix) must be dominating the computation time. The communication benchmarks (appendix I) indicate that the collection of the partial Fock matrices is rapid, so that the bottleneck must be the diagonalization. For this test case (n= 24) this is not surprising. As discussed in the SCF section, when the number of nodes (m) is compar-

able with the number of basis functions (n), the parallel Fock matrix formation and the serial diagonalization steps have the same computational complexity. Although for much larger problem sizes (n >> m) the SCF speedup should be good, it is clearly necessary to parallelize the diagonalization step in order to allow efficient use of large processor arrays.

The third table gives the percentage differences in the timings for the fastest and slowest processors indicating the quality of the load balancing. The results are quite good especially considering the small size of the test case.

Figure 18 gives the same results for the 13 basis function H_2O test case. The results are pretty much the same as those for C_2H_2 except that the smaller size of the problem leads to poorer speedups and load balancing on large numbers of processors. A surprising and rather curious result is the slight super-linear (i.e better than ideal) speedups seen for the first few numbers of processors. Since these speedups have been calculated relative to single node timings this cannot be the result of poor load balancing in the reference calculation (as was the case for the C_2H_2 benchmark). At this time this anomalous result has not been definitely attributed to a specific cause. A careful study of the algorithm has found nothing that would preferentially degrade the single processor performance.. This indicates that some the problem arises from some feature of the node hardware. A likely candidate is associated with how the processor handles large data arrays. The size of the data sets on each node is inversely proportional to the number of nodes participating in the calculation. Hence, if references into very large data arrays are at all slower than references into small arrays, the algorithm's performance would be degraded when running on fewer processors. The fact that

memory access speed is inversely proportional to memory size has been used as an argument for the possibility of super-linear speedups on parallel computers.[49] However, more extensive benchmarks of the Hypercube would be needed to verify that this is the phenomena being observed.

DIRECT SCF

The benchmark results for parallel direct SCF calculations on C_2H_2 and H_2O are given in figures 19 and 20. As before, these data are in three tables giving total run times, speedups, and percentage load imbalance. Each of these tables has two data columns. The first gives results for the complete calculation while the second contains results for a single iteration the parallelized portion of the program. The parallelized portion involves both the integral evaluation and the formation of the partial Fock matrix. As before, the timings given are for the slowest processor. Note that the reference calculation for C_2H_2 was run on three processors (rather than one) due to time limitations; memory is not a limitation for direct SCF calculations.

As might be expected from the previous results, the speedups are very good. The parallelized portions show nearly perfect speedups over the entire range of node configurations. The speedups for the complete calculation is degraded on large numbers of processors (particularly for the smaller test case) as the serial diagonalization step grows to dominate the computation time. This problem can only be alleviated by parallelizing the diagonalization step. Once again the results show slight ($< 5\%$) super-linear speedups for small numbers of processors.

POST-SCF BENCHMARKS

This section describes benchmark results for the full Moller Plesset energy calculations. This procedure involves four steps: integral evaluation, SCF, integral transformation, and the $E^{(2)}$ calculation. Benchmarks calculations were run for both the C_2H_2 and H_2O test cases using both of the proposed Moller Plesset algorithms. These results are given in figures 21-24 in the same format as the results in the previous two sections.

The integral and SCF results are very similar to those presented in the previous sections. As anticipated the speedups and load balancing for these steps is worse due to the more complicated integral distribution constraints imposed by the transformation step. The particularly poor speedup for the SCF step is due to the fact that a more lengthy matrix diagonalization procedure is required since the transformation requires the full set of eigenvectors.

The speedup results for the integral transformation step are quite good. For the larger C_2H_2 test case the speedups are ideal (or even slightly super-linear) for the entire range of processor configurations. The reason for the super-linear results are well understood. It is a consequence of the fact that interprocessor messages are limited to 16 Kbytes on the Hypercube. When small numbers of processors are involved in the transformation of a large test case, many integral blocks must be assigned to each node. For sufficiently large cases, the communication of the partially transformed integrals in the final two steps of the transformation cannot be done in one message passing step. Instead, the integrals must be passed in several communication steps requiring extra overhead for the packing and sending of the message packets. Since this multi-step communication is necessary only when a small number of processors are being used, these cases will have degraded performance.

This suggests that the parallel transformation program does not yield good few-processor reference timings. Instead, the speedups should be calculated relative to a transformation program better suited for single processor operation. This strategy was used for the more extensive transformation benchmarks given in the next section.

As described in the previous chapter two different Moller Plesset algorithms were investigated. One (type 1) involved interprocessor communication in the formation of the super-integrals. The other (type 2) required no communication at the expense of an extra twofold redundancy in the number of integrals evaluated and transformed. The type 1 results are given in figures 21 and 23 and the type 2 results in figures 22 and 24. These results show that the type 2 algorithm is 5-10 times faster than the type 1. Of course the second order Moller Plesset calculation requires so little time that the extra time spent in the integral evaluation and transformation steps far outweighs any gain by using the type 2 algorithm. However, for higher orders of perturbation theory which have greater computational complexities than the transformation the type 2 algorithm should be overall more efficient.

Note that the relatively poor speedups exhibited by the Moller Plesset program are the result of the limited number of occupied orbitals in the two test cases. Since the $E^{(2)}$ expression involves only integrals of the form $(ir|js)$ with i,j occupied and r,s unoccupied, the speedup is largely limited by the number of occupied orbitals in the test case.

Load balancing data is not listed for the transformation and Moller Plesset steps since these procedures involve explicit synchronization of the processing nodes. Thus, overall load balancing would be difficult to measure.

TRANSFORMATION BENCHMARKS

The reason for developing these parallel quantum chemistry programs was to be able ultimately to study very large chemical problems on the massively parallel computers soon to be available. For such problems the computational time will truly be dominated by the most computationally complex steps. Hence it is important to benchmark carefully those steps which will become future bottlenecks. For this reason an extensive series of benchmark calculations were carried out on the two electron integral transformation. In order to save the time and memory space required by the integral and SCF calculations, the transformation was carried out on a set of "dummy" integrals which were a simple function of the AO indices. The transformation matrix C was simply a matrix with all entries equal to 0.5 to simplify the checking of the results. Figure 25 summarizes the timings for various meshes and numbers of basis functions.

In order to determine accurate speedups, a serial transformation program was written. Since this program required too much memory to run large test cases on a single Hypercube node, it was implemented on an IBM 3081 K at the University of California, Berkeley. A series of benchmark calculations revealed that the mainframe executed the transformation 103.5 times faster than the Hypercube node. Thus, the speedup on k processors is defined

$$S_k = \frac{t_{ref}}{t_k} X \ 103.5$$

where t_k is the computer time on k nodes and t_{ref} is the time for the serial calculation on the IBM.

The speedups calculated in this way are given in figures 26 and 27. Figure 26 gives speedup curves for those basis sets sizes where the load evenly

divides onto the processor grid. The results for $n=24$ and $n=32$ are nearly ideal. The curve for $n=16$ shows some falloff, but the efficiency is still greater than 90% on all 32 processors. For the smallest case, $n=8$ there is significant falloff indicating that the problem is so small that the communication is becoming a bottleneck.

Figure 27 shows speedup curves for several problem sizes where even load distribution is not possible for some of the meshes. Since the overall speed of the transformation is limited by the most heavily loaded processor, these results are not as good as those for evenly loaded problem sizes. As $n=24$ is an evenly balanced case, $n=23$ and $n=25$ are two extremes in poor load balancing. For $n=23$ a few processors have less work than the majority, so the speedup is not seriously degraded. However, for $n=25$ only a few processors have extra work, but the remaining majority must wait for these to catch up, leading to a more severe performance degradation. Nevertheless, these results are still quite good considering the relatively small size of the test cases.

SUMMARY

Despite the constraints on problem size due to the limited memory on each node, good performance speedups were found for the integral evaluation and transformation steps as well as the parallel portions of the SCF. The SCF benchmarks clearly indicate the need to parallelize the diagonalization of the Fock matrix. Although the test cases were too small to yield reliable benchmarks for the Moller Plesset step, the speedup on smaller processor meshes is encouraging.

Chapter IV: Conclusions and Future Directions

The work described in this thesis represents a first step in the development of a general system of quantum chemistry programs able to exploit the capabilities of massively parallel computers. Just as the development of efficient serial algorithms for quantum chemistry has taken many years, the remaining steps will require the efforts of many researchers over a number of years. Much of the remaining work has been alluded to in the preceding chapters. The following paragraphs will describe some of the continuing work currently being pursued by the author and his collaborators.

As described in the previous section, most of the programs implemented should give good performance on the parallel computers available in the foreseeable future. A notable exception is the SCF step where the diagonalization step can become a bottleneck on large processor grids. In order to rectify this situation work has begun in implementing the parallel diagonalization algorithm described in this thesis.

A second direction of current work is to complete the implementation of the third and fourth order Moller Plesset energy corrections. Once implemented, these methods will allow truly "state of the art" calculations to be carried out in parallel.

Although not currently under development, the next logical step will be the implementation of a general configuration interaction program. This method has a number of advantages over Moller Plesset perturbation theory, probably the most important being that it yields a variational upper bound on

the total energy. Additionally, the general methodology of forming and diagonalizing a large Hamiltonian matrix is applicable to a large number of problems in the physical sciences.

Finally, a straight forward but practically useful task will be the implementation of programs to compute the first and second derivatives (with respect to nuclear coordinates) of these various total energies (SCF, MPPT, and CI). These procedures are well worked out for serial computers, and the bottleneck steps involve the computation of large integral lists and the solution of large systems of linear equations, both of which should easily parallelize.

Although the algorithms described in this thesis have been designed to work efficiently on distributed memory parallel computers, they are sufficiently general that they should provide good performance on a wide range of parallel architectures. Work is currently under way to get these programs running on two vastly different parallel computers.

The first of these is a ten processor Elxsi 6400 located at Sandia National Laboratories. Although this computer has a shared memory architecture and hence no explicit interprocessor communication channels, communication can be mimicked by routines using the shared memory. Work is under way writing Hypercube compatible communication routines. These routines will require computational overhead to mimic effectively communication channels, but this should not adversely affect the performance of the algorithms since they were specifically designed for systems with relatively slow interprocessor communication. This suggests that all parallel algorithms should be targeted for distributed memory computers since shared memory computers can effectively mimic distributed memory architectures, while the converse is

not true.

A second type of parallel computer on to which these programs are being implemented is not a tightly coupled set of homogeneous processors as are the Hypercube or Elxsi. Instead, this "computer" is a large group of workstations linked together by a local area network. Such collections of fairly powerful workstations (each comparable to a DEC VAX 11/780) linked by communication channel to each other and shared resources (such as printers and disk drives) are becoming increasing popular and will soon be a ubiquitous presence in businesses and research institutions. Although these workstations are individually too small to perform large scale scientific computations, together they represent an enormous computing resource (especially since they are often idle during evenings and weekends). For example the network of ~ 200 SUN 3/50 workstations currently in place at the University of California at Berkeley is, as a unit, several times the power of the Cray X-MP at the central computing facility.

All that is required to use these large networks as a single distributed memory computer is to write a series of Hypercube compatible communication routines using the existing network protocols. The relatively slow network communication speeds should not seriously degrade the performance of the algorithms described here. A set of preliminary routines have been written by Curtis Janssen at Berkeley. The speedup curve for a 26 basis function integral and SCF calculation running on 1-8 SUN 3/50 workstations is given in figure 28. Benchmark calculations will be carried out on much larger arrays of workstations as soon as a more efficient set of communication routines is complete.

In addition to the development of parallel algorithms for the standard

quantum chemical methods, an important direction of study is the development of entirely new approaches to molecular quantum mechanics which involve inherently parallel techniques. One promising method which has been developed during the past decade is Quantum Monte Carlo (QMC).[50] The basic idea behind QMC is to recast the Schrodinger equation as a diffusion equation which is then solved by a stochastic algorithm. This algorithm involves carrying out standard Monte Carlo updates on a large ensemble of possible configurations of the system (i.e. trial wavefunctions). After a certain number of generations, configurations are either discarded or replicated on the basis of a Boltzman weighting of their total energy. (This is called the branching step.) Then the Monte Carlo procedure is again carried out on the new ensemble of configurations. In this way, the ground state wavefunction eventually evolves.

Since this algorithm involves the concurrent simulation of a large ensemble of configurations it is ideal for parallelization. Each processor is assigned a configuration or set of configurations for the Monte Carlo updates. Interprocessor communication is required only during the branching step. Unfortunately, there are still some problems using QMC on fermion systems related to the selection and convergence of the wavefunction's nodal structure. However, these problems are being addressed so that QMC should be a very promising method for parallel implementation in the coming years.

Another potentially promising method involves the solution of the SCF equations using pseudospectral methods originally developed for hydrodynamic simulations. [51,52] The basis strategy is to construct different parts of the Hamiltonian matrix (in this case the Fock operator) in different representations. Certain parts of the Hamiltonian are best calculated in the spectral

(i.e. basis set) representation. These terms include the kinetic energy terms involving derivatives with respect to position. Other terms including the electron-electron repulsion terms are best constructed in the spatial representation (i.e. on a three dimensional grid). The collocation method is used to transform between these two representations.

The net effect of this approach is that the number of two electron integrals is reduced from n^4 to n^3 . This is potentially a big advantage for the development of parallel implementations since the amount of memory on each processor is one of the chief limitations on the size of problems which can be studied.

Of course a lot of research on parallel processing is being carried out by the computer science community. Of particular interest to computational physical scientists is the work on software to parallelize automatically serial programs. Ideally programs should not have to be specifically tailored to utilize efficiently a particular computer. Instead the compiler should be able to restructure the program for optimal performance regardless of whether the architecture is scalar, vector or parallel.

Vector compilers have been under development for nearly a decade and are now fairly adept at reordering loop structures to optimize vector performance. By comparison, automatic parallelization compilers are still quite new and limited in capability. Those currently available seem capable of carrying out efficiently very fine grained parallelization. More specifically, these systems distribute the program on an instruction by instruction basis, for example, distributing the successive cycles though a do-loop. While this approach has proven successful so far, the fine grained nature of the parallelism requires very high interprocessor communication rates and hence will be lim-

ited to relatively small numbers of processors.

In order to achieve the very large performance enhancements promised by massively parallel computers much coarser grained parallelism will have to be used. However, the automatic detection and exploitation of coarse grained parallelism is very difficult. The problem is that this requires knowledge of the control structure and data dependencies of the program, and such information is very hard to extract from programs written in standard scientific programming languages such as FORTRAN. Unfortunately this difficulty is not due to superficial features, but rather due to the fundamental structure of these languages.

A particularly problematic feature of languages like FORTRAN is that all variables point to memory locations ("pass by pointer"). While this has the advantage of saving memory, it means that the input values to functions and subroutines can (and often are) modified by the routine. Such modifications are called side effects and they greatly complicate the problem of parallelization. For example, consider the following segment of code:

$$D = func1(A, B)$$
$$E = func2(A, C)$$

Potentially $func1$ and $func2$ could be evaluated in parallel. However since $func1$ may modify A, extensive analysis of $func1$ is necessary before parallelization can be implemented.

These difficulties are sufficiently serious that it is unlikely (at least in the near future) that a fully automated compiler will be available that is capable of restructuring efficiently a program for a distributed memory computer. Of course many software tools will be available to aid in the development of parallel programs, but the burden of determining how to distribute the

program will still be on the programmer.

In order to overcome some of the difficulties associated with standard scientific programming languages, computer scientists have developed a new type of programming language known as functional languages.[53] In these languages the programmer works entirely with instructions that act like mathematical functions, taking particular inputs and returning outputs. An example of a simple function is $sum(A,B)$ which returns $A+B$. Note that A and B are unmodified (i.e. there are no side effects). From simple operations such as *sum* the programmer constructs more and more complex functions until ultimately the program itself is defined in terms of these high level functions. For example the following is a hypothetical SCF program:

$$scf\_energy = sum(elect\_energy(nuc\_coords, basis), nucl\_energy(nuc\_coords))$$

where the total SCF energy is calculated as the sum of two functions calculating the electronic and nuclear repulsion terms.

Such an approach has a number of useful properties. It produces well-structured, easy to read programs and it easily allows the construction of large programs from sets of smaller programs. Further, functional programs are in principle easily parallelized. The reason for this is that since the functions have no side effects, all terms in a given expression can be evaluated concurrently. In the example given above, the functions *elect_energy* and *nucl_energy* can be evaluated simultaneously.

Despite these advantages, functional languages are somewhat difficult to use and are inefficient on standard computer architectures. Hence, they are still not universally advocated. Exactly how much of the task of parallelization will eventually belong to the programmer is still unclear; however, it is likely that efficiently programming parallel computers may always be more

difficult than serial programming. Nevertheless, if large performance increases are needed, this extra effort is justified.

A final question that should be addressed in this thesis is whether there are ultimate limits on the speed and accuracy of scientific computation. This question is more subtle than that of the limitations on processing speeds. Parallel computers can in principle be made arbitrarily fast by linking together arbitrarily large numbers of processors. Hence, the real limit is not the maximum potential speed of the parallel computers, but instead the ultimate degree to which scientific problems can be parallelized. Of course this largely depends on the scientific problem at hand. Therefore, for simplicity consider the numerical computation of the dynamics of a many-body system. Since this computation is simply a direct simulation of nature, the limits on its parallelizability are tied the deeper, more general question: to what extent are natural processes parallelizable? That is, to what extent can nature be decoupled into separate processes interacting only locally?

The difficulty with this decoupling is that at the scales nature is usually simulated, particles are assumed to exert instantaneous action at a distance. Thus, for a system spatially mapped onto a grid of processors the motion of particles on distant processors will to some degree be coupled. This coupling will require interprocessor communication putting tight constraints on the number of processors which can be efficiently used.

All natural forces attenuate with distance so that there is a cutoff distance beyond which particles can be considered non-interacting. Therefore the limit on the numbers of processors which can be efficiently used is determined by the ratio of the distance of significant interactions to the size of the system. For example, if the system being simulated is a relatively small clus-

ter of very massive particles, all particle motions will be tightly coupled so that the system will not be amenable to massive parallelism. In contrast, systems with no action at a distance such as continuum fluids and "billiard-ball" models are ideal for massive parallelism. For such systems the only limitation will be the communication required to calculate global system properties such as energy, average momentum, and so on.

Since interprocessor communication limits the performance of parallel computers, one strategy would be to shorten the message size and data routes. This is the motivation for radical new computer architectures which involve thousands of tiny few bit processors. Since the processors are very small many can be fit onto a single integrated circuit so that the communication channels are very short. Moreover, the processors are designed such that the largest useful messages are only a few bits.

In order to use such machines efficiently, a very non-traditional approach to scientific simulation is required.[54] Rather than stylize large scale dynamics into complex partial differential equations to be solved by numerical schemes, the microscopic dynamics are encoded into very simple (few bit) rules describing the local interactions of particles or small spatial regions. The large scale behavior of the system is determined by studying very large aggregates of the interacting sub-regions. Since the interaction rules are simple and local, such algorithms are easily mapped onto these novel computer architectures.

The first major application of this strategy has occurred only in the past year when Pomeau and coworkers[55] discovered that the large scale dynamics of two dimensional incompressible fluid flow can be reasonably modeled by a large ensemble of simply interacting point particles moving on a hexagonal

mesh. This so-called hexagonal lattice gas model is straight forward to parallelize and has already been efficiently implemented on a 65000 processor Connection Machine.[56] Despite its recent development this model has stirred a great deal of scientific research and raised many fundamental questions about hydrodynamics and computational simulations.[57] Work is currently underway to extend this approach to more complex systems including multicomponent fluids[58] and plasmas.[59]

Despite this success, it is still not clear whether these methods will be applicable to strongly interacting, noncontinuum systems. Of course the ultimate simulation of nature would be carried out at such a small scale that instantaneous action at a distance would not occur--particles would interact directly by exchange of virtual particles. However at these scales the particles themselves would be nonlocal wavepackets, leading to even deeper questions about the simulation of fundamental interactions. Hence, the question of whether there exist ultimate limits on the scale and accuracy of scientific simulations is not yet answerable; however, it is clear that the attempt to reach this limit will uncover clues to the ultimate goal of science, an understanding of the fundamental workings of nature.

Appendix 1: Hypercube Benchmarks

A series of benchmark calculations were carried out to test the processing speed and interprocessor communication rate of the Intel Hypercube. The following table gives the timings for various operations on double precision real variables. These benchmarks were run on the processing nodes using the system *clock* function to time the operations. The timing values are in millisec and indicate the amount of time to carry 100000 repetitions of the operation. In this table R indicates a double precision real variable, and the final table entry is the overhead time for a do-loop of length 100000.

Floating Point Benchmarks		
Operation	Millisec per 10^5 Ops	Ops per second
$R*R$	7728	12940
$R*R*R$	10304	9705
$R+R$	6624	15087
$\frac{R}{R}$	8144	12278
$\sqrt{R}$	7440	13440
$exp(R)$	37056	2699
do-loop	2528	--

With do-loop overhead taken into account the processing speed of each node is 19230 double precision multiplications per second. Hence if all 32 nodes were used with 100 percent efficiency, the overall processing speed of the Hypercube would be approximately .62 MFLOPS (million floating point operations per second) or about the speed of 6 DEC VAX 11/780's.

Determining the interprocessor communication rates is a more complicated procedure. Since the clocks on the different nodes are not necessarily synchronized, it is not possible to time directly the communication of messages between nodes. Instead, it is necessary to time the cycle of a message from a node to its neighbor and then back to the original node again. Since this round trip involves two complete message passing steps, the one way communication rate can be calculated.

The following table gives the round trip communication times and the derived one way message passing rates.

Node-to-Node Communication Rates			
Message Size (bytes)	Iterations	Time (msec)	Kbytes per sec
4	100	0.90	0.89
40	100	1.07	7.46
1000	300	4.32	138.9
1040	300	5.52	113.0
2040	300	5.87	208.4
2080	300	8.26	151.2
4080	300	12.26	199.7
4120	300	13.20	187.3
8160	300	322.64	216.3
8200	300	24.35	202.0
12280	300	33.39	220.7
12320	300	35.01	211.2
16000	200	42.91	223.7

The communication rate is plotted against the message size in figure 29. The asymptotic interprocessor communication rate is 225 Kbytes per second for one-way, nearest neighbor message passing. The node to host communication rate was calculated in a similar way and found to be 75 Kbytes per second. Further, these results indicate that the net communication rate is strongly dependent on the message size and that the messages are broadcast in 1 kbyte chunks. Hence, when developing algorithms for the Hypercube, communication overhead can be minimized by avoiding small messages or by bundling them together into larger blocks. Comparing the node processing speed of 19230 floating point operations per second with the maximum communication rate indicates that about 1.5 double precision numbers can be passed between processors in the time that one operation can be carried out.

Appendix 2: Benchmark Test Cases

The parameters for the two benchmark test molecules are given in the following tables. Note that the basis sets are fully uncontracted to simplify load balancing.

C_2H_2

Geometry	
C– C	1.50 Bohr
C– H	0.75 Bohr

Energies	
SCF	– 112.464174140 Hartrees
Moller Plesset ($E^{(2)}$)	– .108571920639 Hartrees

Basis Sets					
Carbon			Hydrogen		
#	Ang. Mom.	Exponent	#	Ang. Mom.	Exponent
1	S	42.4974	1	S	2.8992
2	S	14.1892	2	S	0.6534
3	S	5.1477	3	S	0.1776
4	S	1.9666			
5	S	0.4962			
6	S	0.1533			
7-9	P	0.54424			

H_2O

Geometry	
O– H	1.8523498 Bohr
H– O– H	104.0330035 degrees

Energies	
SCF	– 63.195575507070 Hartrees
Moller Plesset ($E^{(2)}$)	– 0.0694749326263 Hartrees

Basis Sets					
Oxygen			Hydrogen		
#	Ang. Mom.	Exponent	#	Ang. Mom.	Exponent
1	S	5.0	1	S	0.5
2	S	1.0	2	S	1.0
3	S	0.5			
4-6	p	0.75			
7-9	P	1.25			

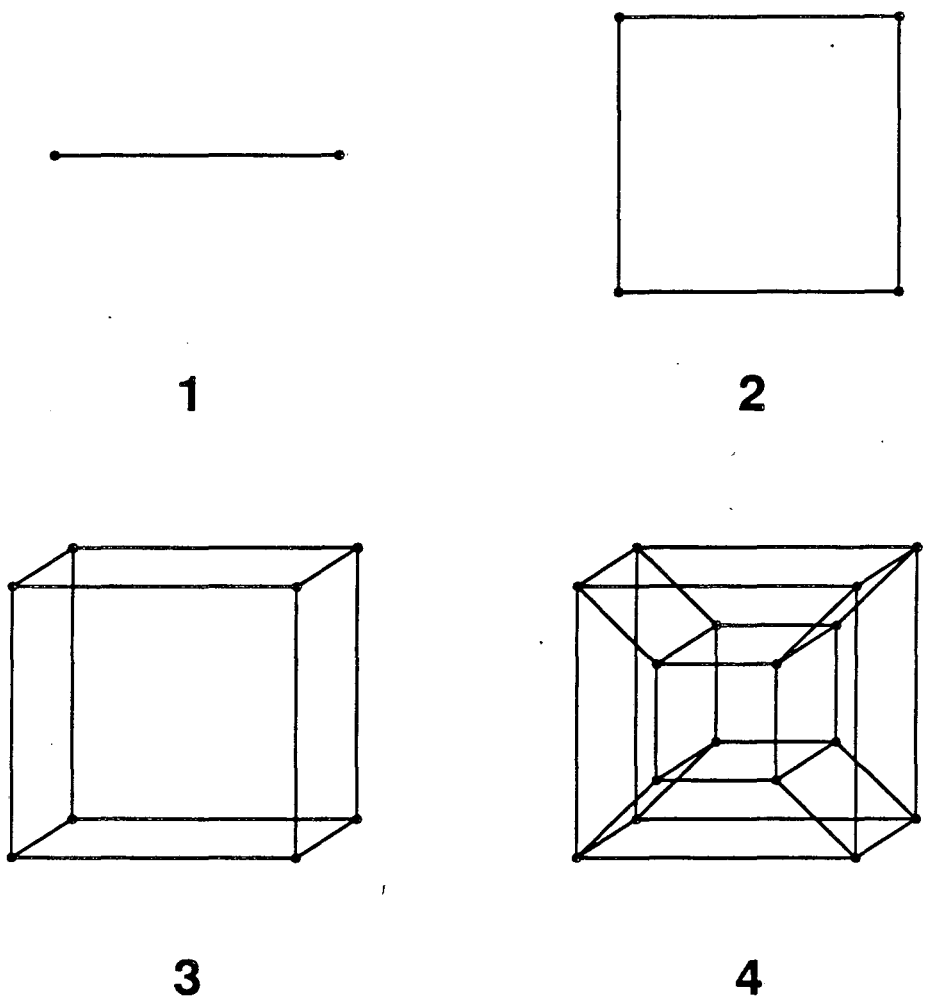


Figure 1. Illustration of hypercubes of dimensions 1,2,3, and 4. Vertices represent processors and edges represent direct interprocessor communication channels.

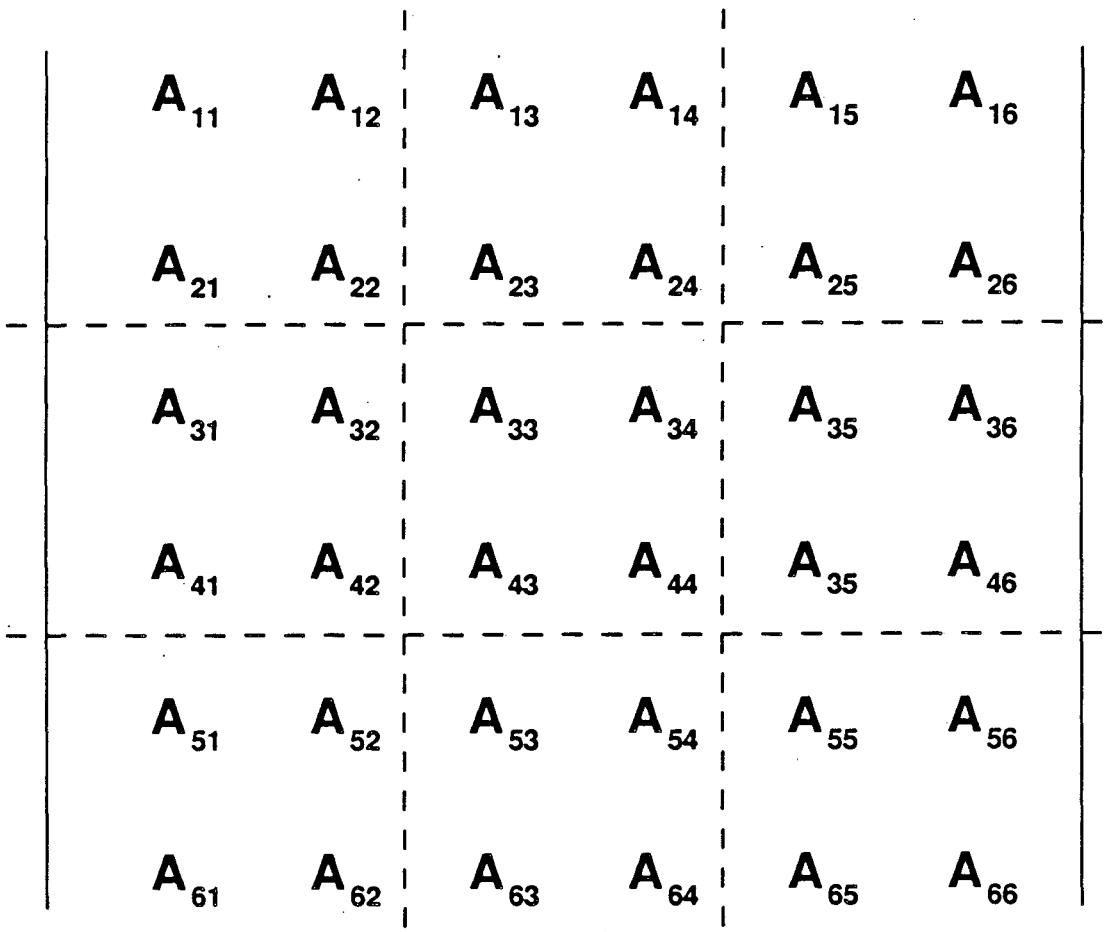


Figure 2. Distribution of matrix elements onto a 3 x 3 grid of processors. Dashed lines indicate indicate processor boundaries.

```
      Program Host
C
C This program reads in a list of numbers, sends them
C to separate nodes for processing, and then sums the
C results together
C
      Integer*4 ci,mtype,length,node,pid,rlength
      Dimension x(32)
C
C Open communication channel
      ci=copen(1)
C
C Read in the number of node processors
      Read *,nproc
C
C Read in nproc numbers
      Read *,(x(i),i=1,nproc)
C
C Send these values to separate processors
      mtype=1
      length=4
      pid=1
      Do 10 i=1,nproc
      node=i-1
      Call sendmsg(ci,mtype,x(i),length,node,pid)
   10 Continue
C
C Receive values back from nodes
      total=0.0
      Do 20 i=1,nproc
      Call recvmsg(ci,mtype,value,length,rlength,node,pid)
      total=total+value
   20 Continue
C
C Print out result
      Print *,'Total =',total
      End
```

Figure 3a. Listing of sample program for the host processor.

```
      Program Node
C
C This program receives a value from the host, squares it
C and sends it back to the host.
C
      Integer*4 host,pid,mtype,length,rlength,ci,copen
C
C Open communications channel
      ci=copen(1)
C
C Receive value from host
      mtype=1
      length=4
      Call recvw(ci,mtype,value,length,rlength,host,pid)
C
C Square this number
      value=value*value
C
C Send new value back to host
      Call sendw(ci,mtype,value,length,host,pid)
C
      End
```

Figure 3b. Listing of sample program for the node processors.

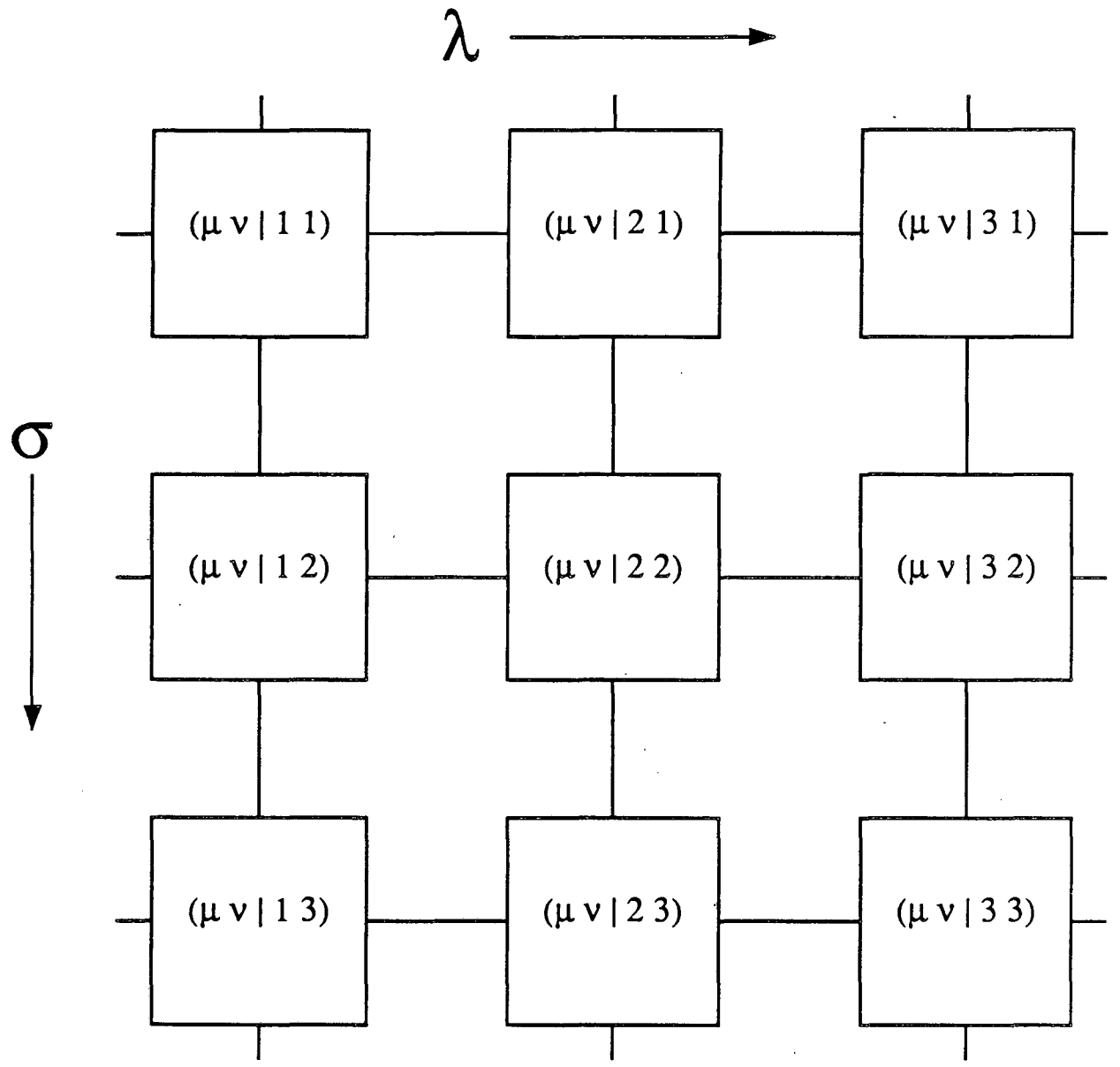


Figure 4. Distribution of two electron integrals for n = 3 on 3 x 3 grid of processors. The Greek indices range over all symmetry distinct values: $\mu,\nu = 1$ to n, with $\mu \geq \nu$.

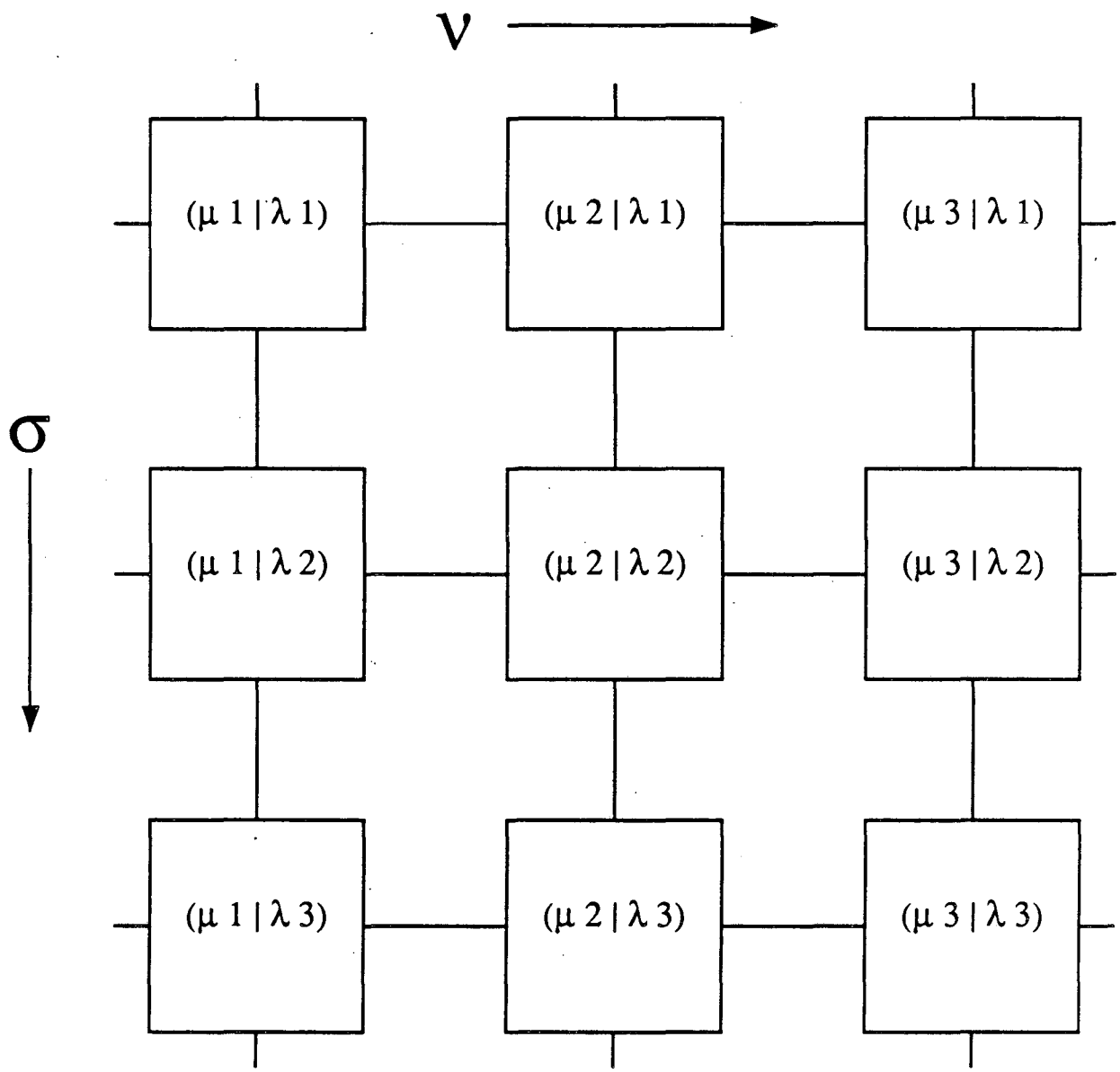


Figure 5. Distribution of two electron integrals for n = 3 on 3 x 3 grid of processors. The Greek indices range over all values: μ,λ = 1 to n.

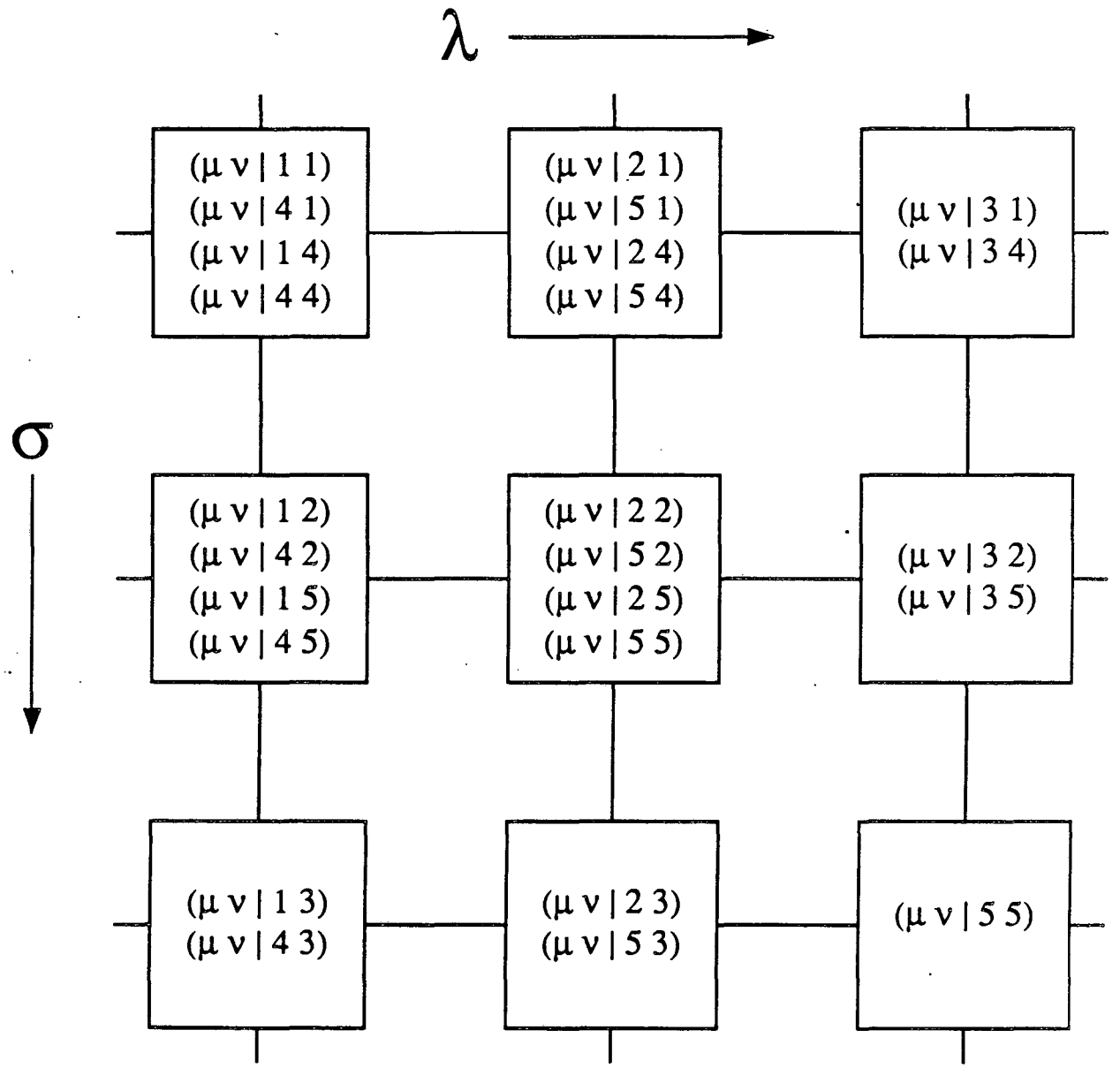


Figure 6. Distribution of two electron integrals for n = 5 on 3 x 3 grid of processors. The Greek indices range over all symmetry distinct values: $\mu,\nu = 1$ to n, with $\mu \geq \nu$.

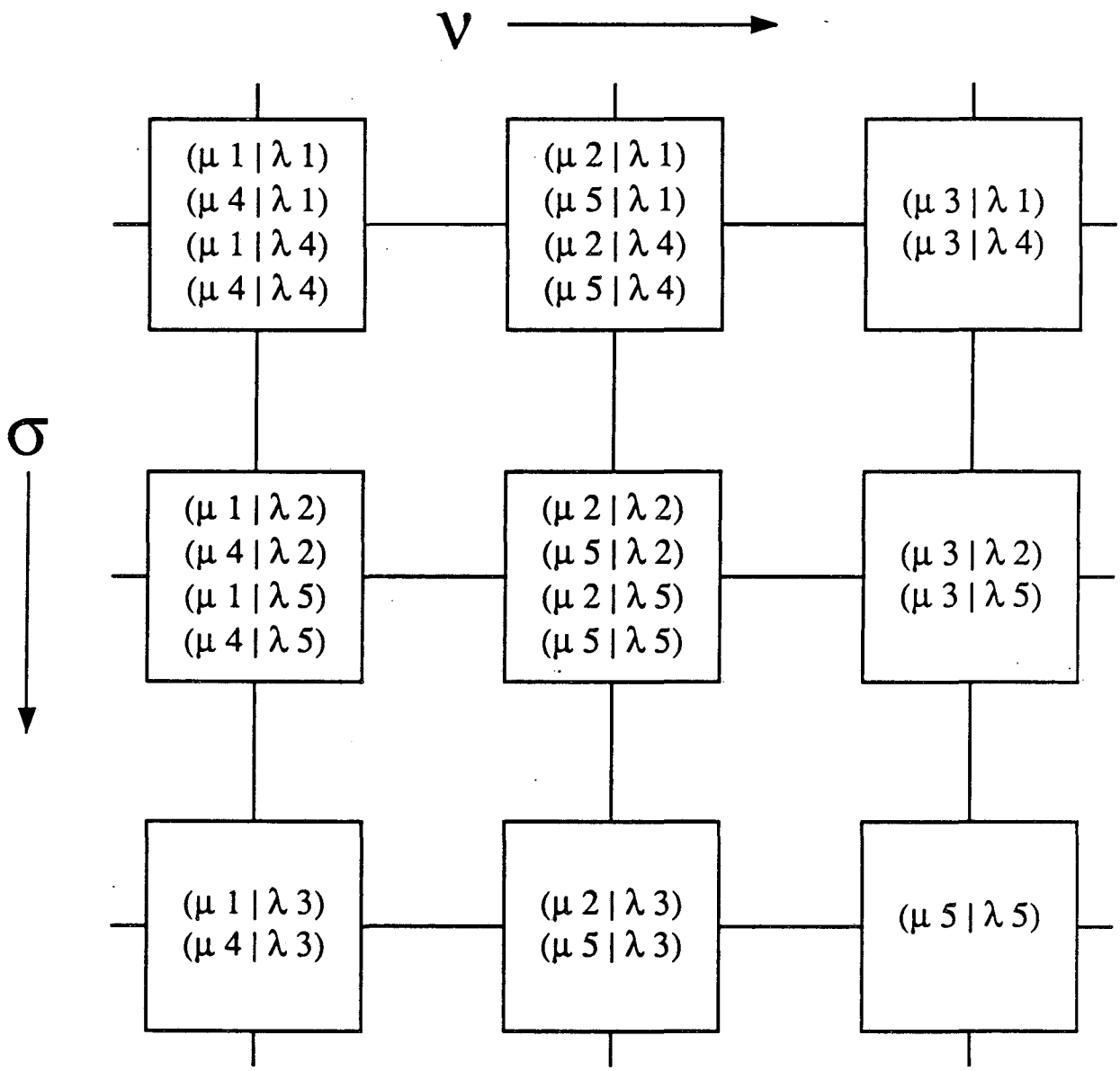


Figure 7. Distribution of two electron integrals for n = 5 on 3 x 3 grid of processors. The Greek indices range over all values: μ,λ = 1 to n.

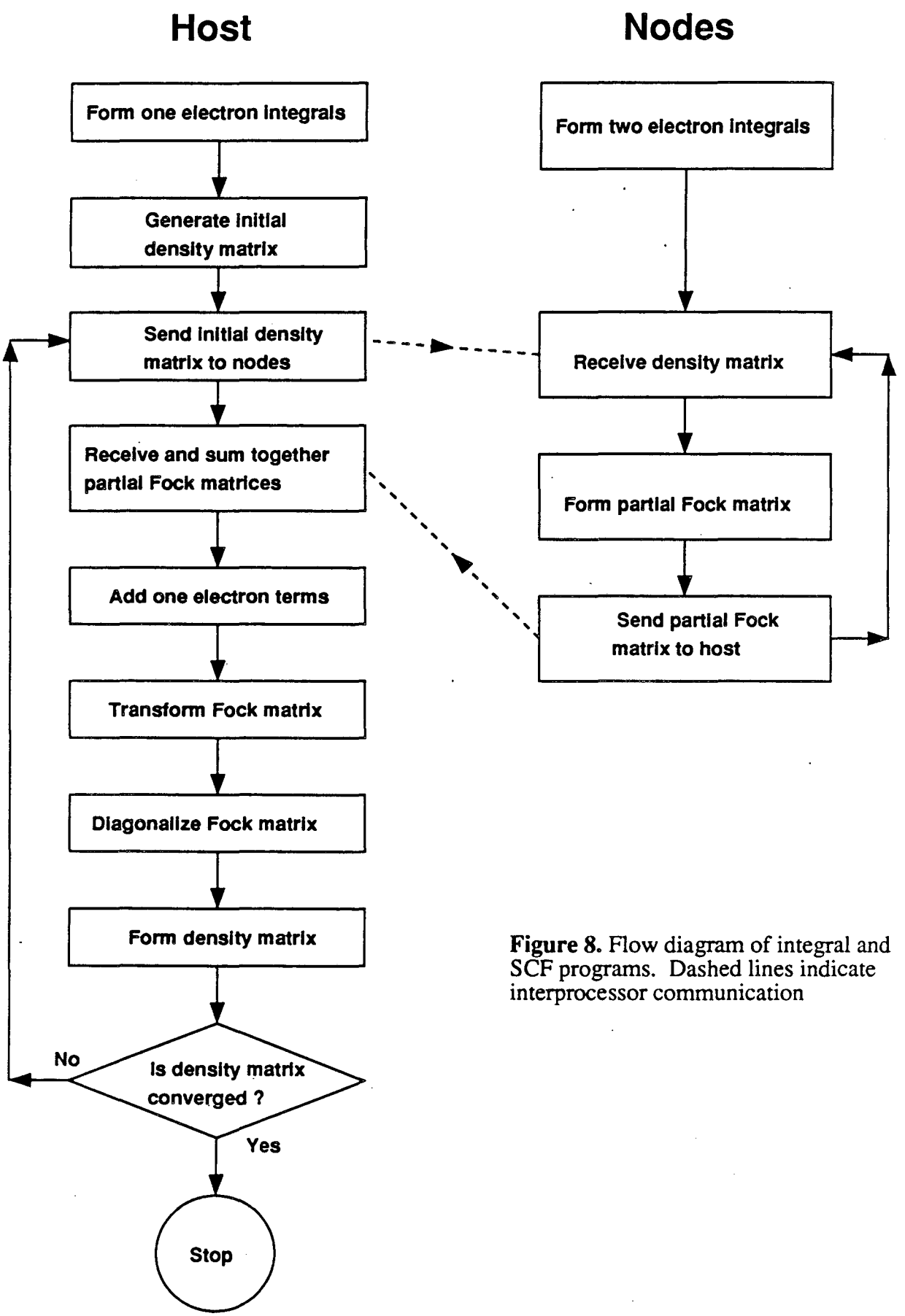


Figure 8. Flow diagram of integral and SCF programs. Dashed lines indicate interprocessor communication

Fock Matrix Contributions									
Integral	F_{aa}	F_{ba}	F_{bb}	F_{ca}	F_{cb}	F_{cc}	F_{da}	F_{db}	F_{dc}
$(aa\|aa)$	$\frac{1}{2}P_{aa}$	-	-	-	-	-	-	-	-
$(ba\|aa)$	P_{ba}	$\frac{1}{2}P_{aa}$	-	-	-	-	-	-	-
$(bb\|aa)$	P_{bb}	$\frac{-1}{2}P_{ba}$	P_{aa}	-	-	-	-	-	-
$(ba\|ba)$	$\frac{-1}{2}P_{bb}$	$\frac{3}{2}P_{ba}$	$\frac{-1}{2}P_{aa}$	-	-	-	-	-	-
$(bb\|ba)$	-	$\frac{1}{2}P_{bb}$	P_{ba}	-	-	-	-	-	-
$(cb\|aa)$	$2P_{cb}$	$\frac{-1}{2}P_{ca}$	-	$\frac{-1}{2}P_{ba}$	P_{aa}	-	-	-	-
$(ca\|ba)$	P_{cb}	$\frac{3}{2}P_{ca}$	-	$\frac{3}{2}P_{ba}$	$\frac{-1}{2}P_{aa}$	-	-	-	-
$(cb\|ba)$	-	$\frac{3}{2}P_{cb}$	$\frac{-1}{2}P_{ca}$	$\frac{-1}{2}P_{bb}$	$\frac{3}{2}P_{aa}$	-	-	-	-
$(ca\|bb)$	-	$\frac{-1}{2}P_{cb}$	$2P_{ca}$	P_{bb}	$\frac{-1}{2}P_{aa}$	-	-	-	-
$(cc\|ba)$	-	P_{cc}	-	$\frac{-1}{2}P_{cb}$	$\frac{-1}{2}P_{ca}$	$2P_{ba}$	-	-	-
$(cb\|ca)$	-	$\frac{-1}{2}P_{cc}$	-	$\frac{3}{2}P_{cb}$	$\frac{3}{2}P_{ca}$	P_{ba}	-	-	-
$(dc\|ba)$	-	$2P_{dc}$	-	$\frac{-1}{2}P_{db}$	$\frac{-1}{2}P_{da}$	-	$\frac{-1}{2}P_{cb}$	$\frac{-1}{2}P_{ca}$	$2P_{ba}$
$(db\|ca)$	-	$\frac{-1}{2}P_{dc}$	-	$2P_{db}$	$\frac{-1}{2}P_{da}$	-	$\frac{-1}{2}P_{cb}$	$2P_{ca}$	$\frac{-1}{2}P_{ba}$
$(da\|cb)$	-	$\frac{-1}{2}P_{dc}$	-	$\frac{-1}{2}P_{db}$	$2P_{da}$	-	$2P_{cb}$	$\frac{-1}{2}P_{ca}$	$\frac{-1}{2}P_{ba}$

Figure 9. Symmetry distinct two electron integral contributions to the Fock matrix. For example the integral (21|11) has contributions to two Fock matrix elements: P_{21} * (21|11) to F_{11} and $\frac{1}{2}P_{11}$ * (21|11) to F_{21} .

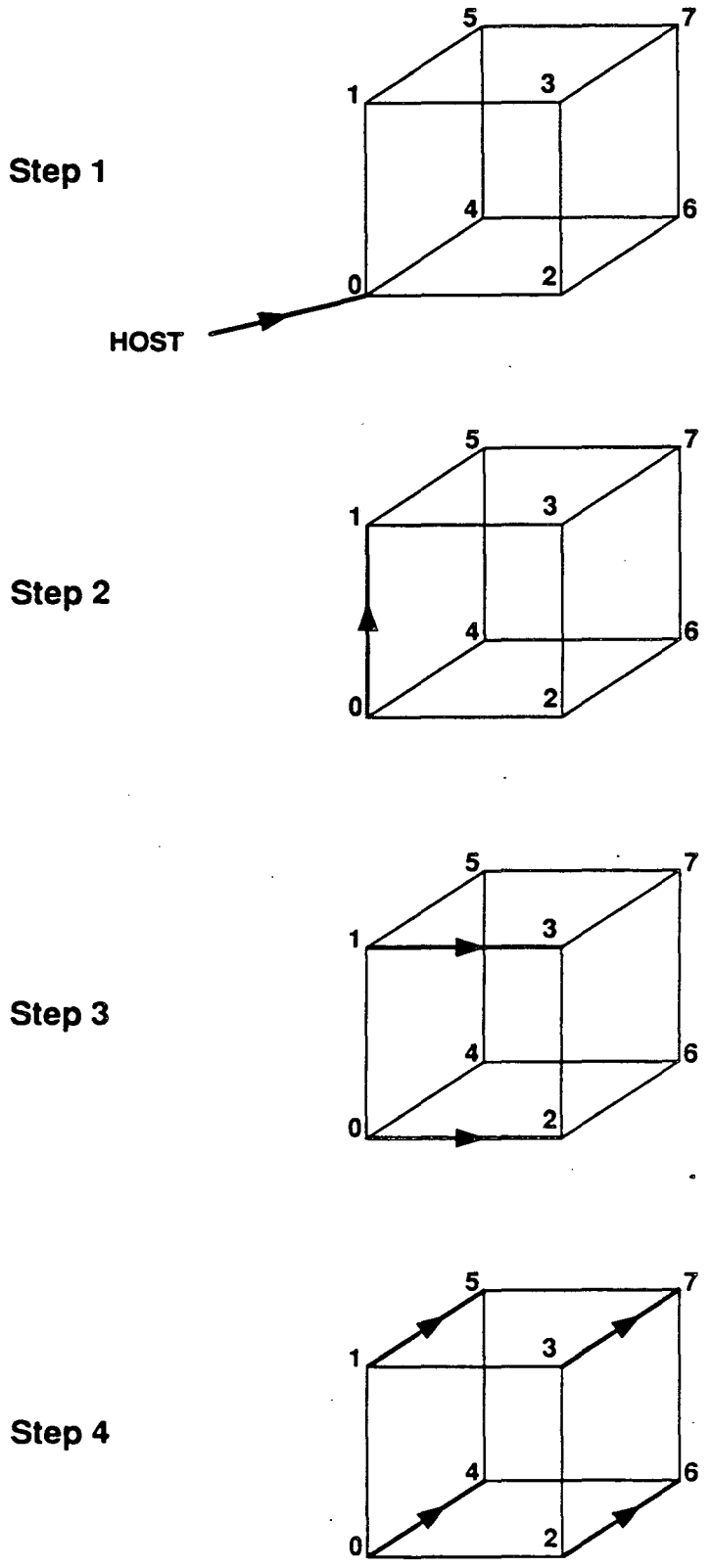
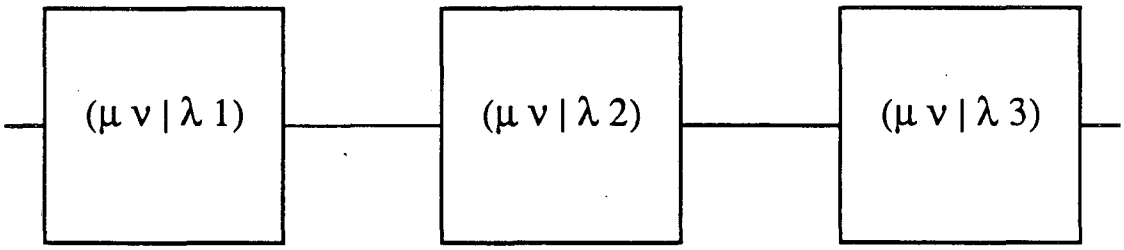
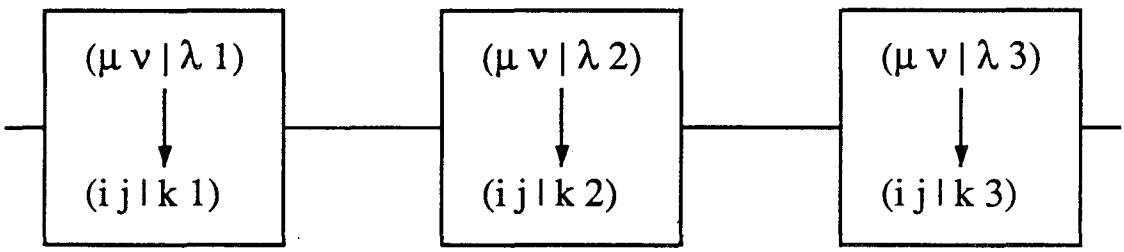


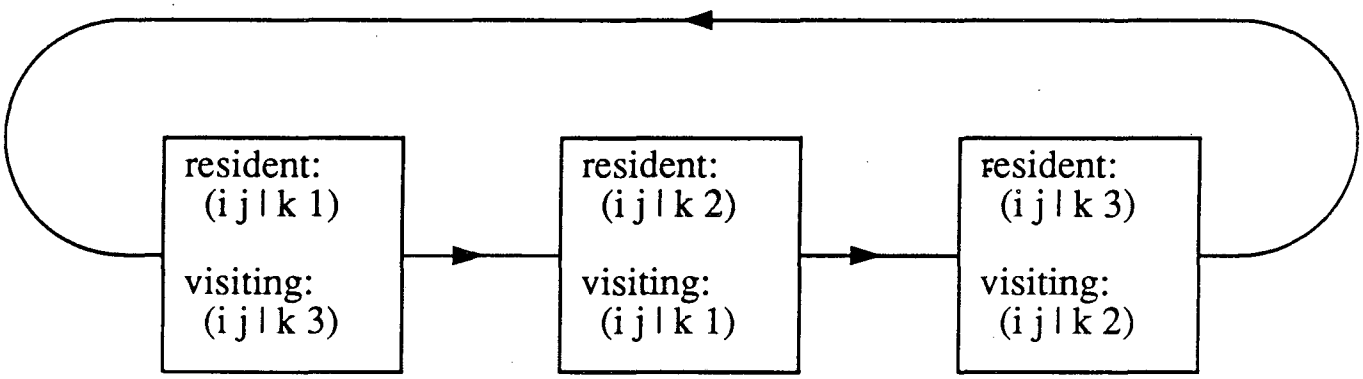
Figure 10. Communication of message from host to all nodes using broadcast tree.



Step 1: Initial distribution of AO integrals.

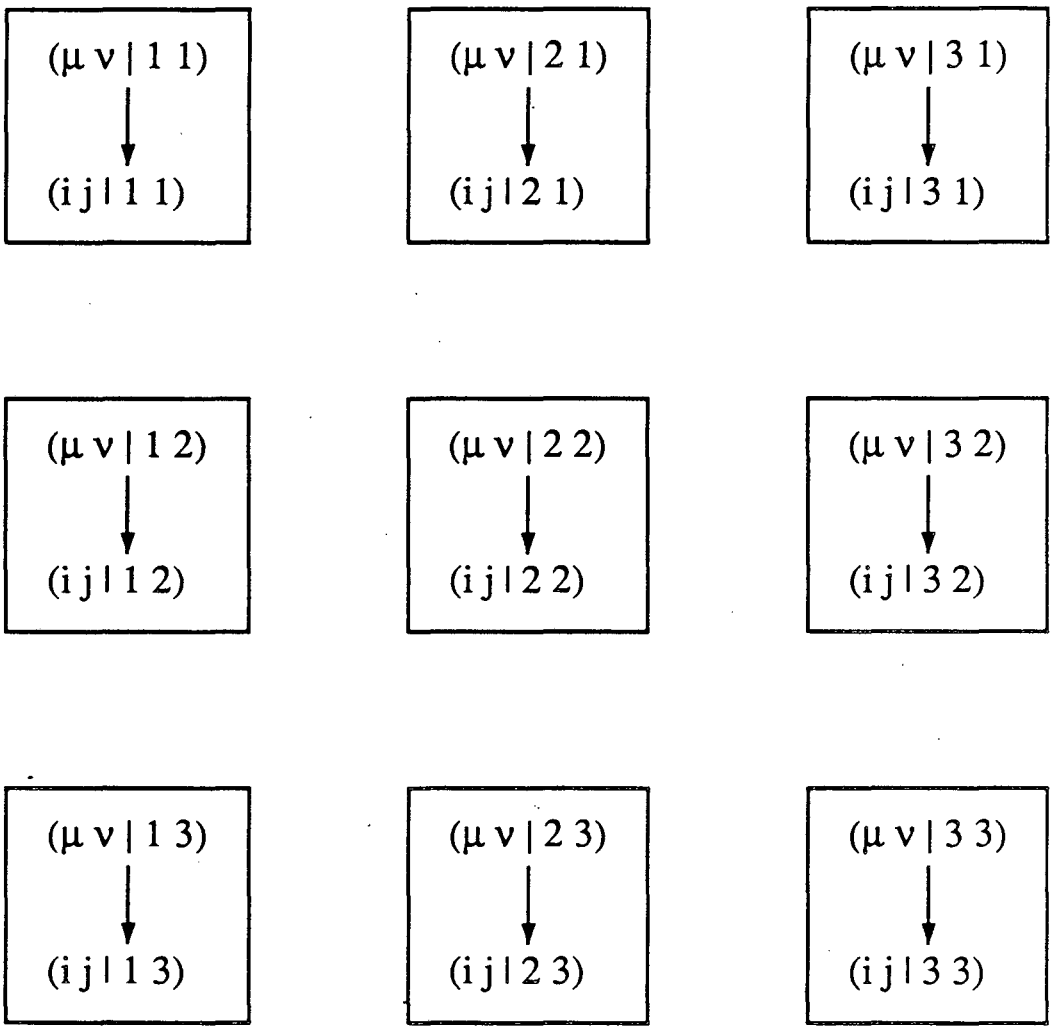


Step 2: In-place transformation of first 3 indices.



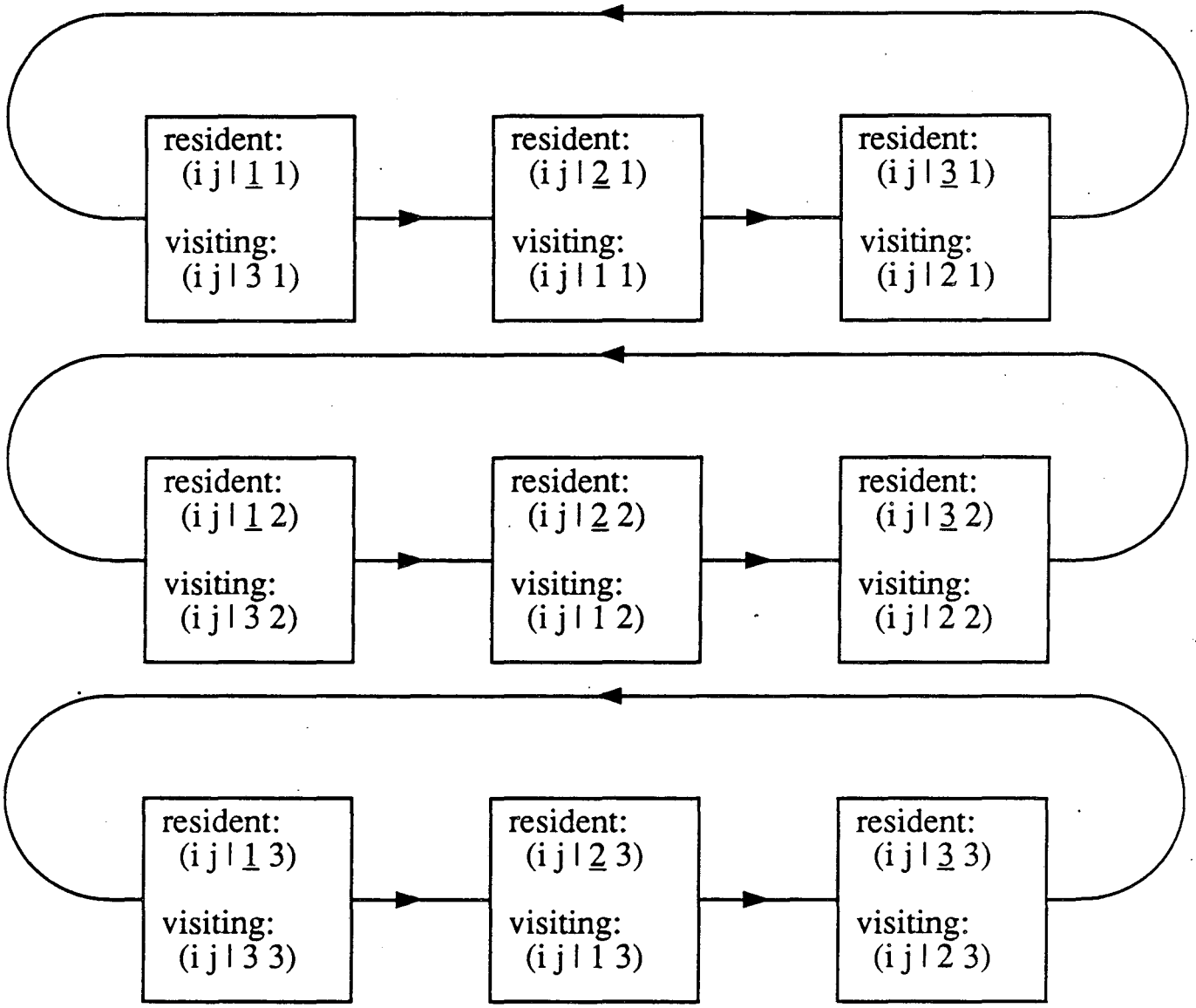
Step 3: Transformation of final index requiring communication around processor ring.

Figure 11. Transformation of three basis function case mapped onto a one dimensional grid of 3 processors.



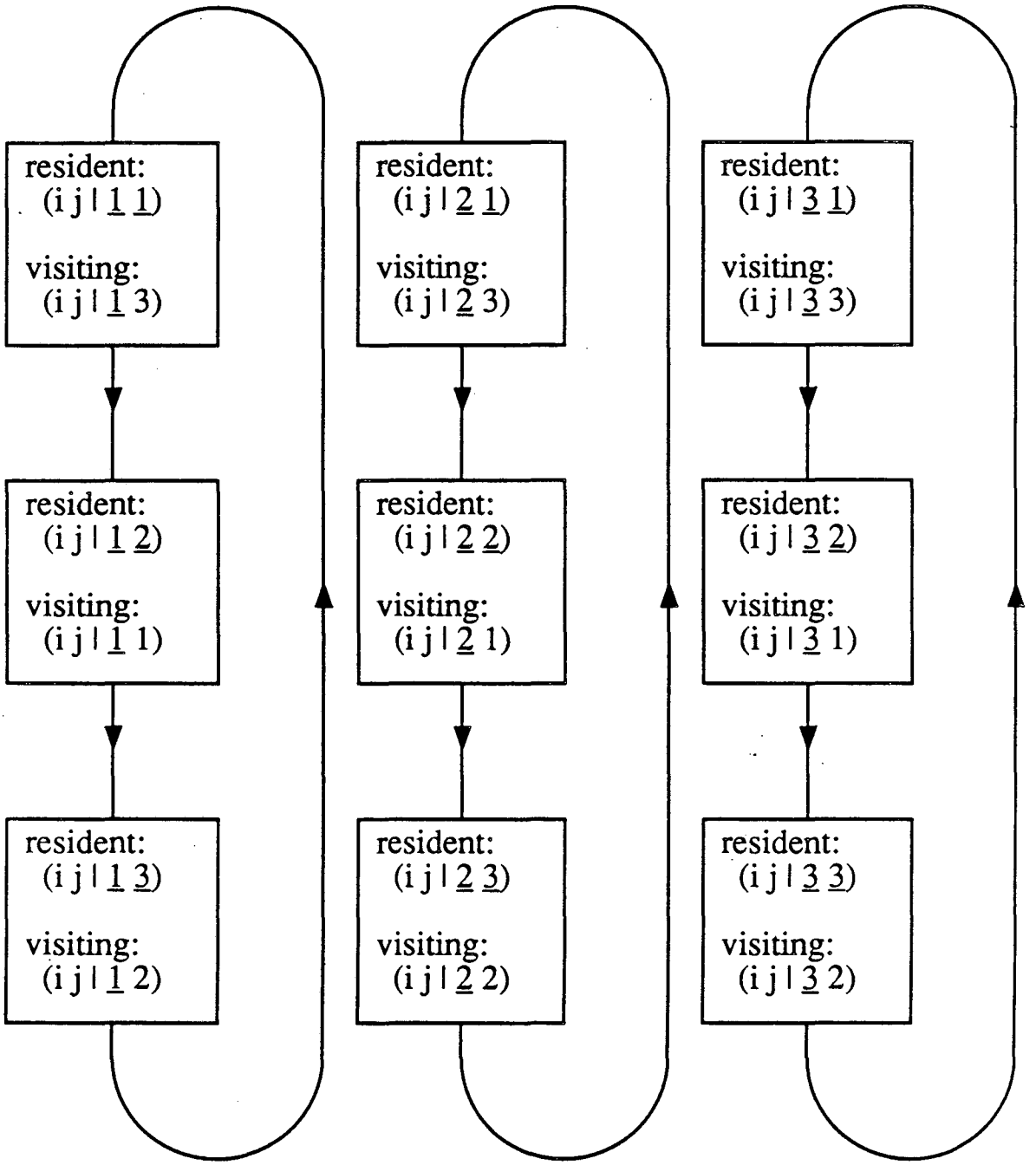
Step 1. In-place transformation of first two indices.

Figure 12. Transformation for n = 3 on 3 x 3 grid of processors.



Step 1. Transformation of third index. Underlined numbers are MO indices.

Figure 13. Transformation for n = 3 on 3 x 3 grid of processors.



Step 1. Transformation of fourth index. Underlined numbers are MO indices.

Figure 14. Transformation for n = 3 on 3 x 3 grid of processors.

```
subroutine trans(xints,buf,c,mylambda,k,nbasis)

comment: Perform the transformation on the index λ , xints
      holds the integrals, buf is a big scratch array, c is the MO coefficient matrix, mylambda is index of the
      resident integral block, k is the index of the MO block to be formed on the processor.

dimension buf(2,nbasis,nbasis), xints(nbasis,nbasis), c(nbasis,nbasis)

comment: initialize the pointers into buf .

 inpoint=2
 outpoint=1

comment: Copy the resident integrals into the first half of buf and zero
       xints.

do μ = 1,nbasis
      do ν = 1,nbasis
            buf(outpoint,μ,ν)=xints(μ,ν)
            xints(μ,ν)=0.0
      end do(ν)
end do(μ)

comment: lambda holds the λ value for the current integral block. Initially set to resident λ.

lambda = mylambda

comment: loop over  nu mber of processors in row (= nbasis). With each pass process the contents of the half
of buf pointed to by outpoint.

do iloop = 1,nbasis

      comment: send the integral block about to be proceed to the neighboring
            processor and initiate receipt of a new integral block.

      call send(buf(outpoint))
      call recv(buf(inpoint))

      comment: sum into xints the contributions from outpoint half of buf.

      do μ = 1,nbasis
            do ν = 1,nbasis
                  xints(μ,ν) = xints(μ,ν) + C(k,lambda)*buf(outpoint,μ,ν)
            end do (ν)
      end do (μ)

      comment: update value of lambda to match incoming buffer.

      lambda = lambda − 1
      if (lambda.eq.0) lambda = nbasis

      comment: Check to make sure new integral buffer has been received.

      call waitchan

      comment: now swap values of outpoint and inpoint.

      call swap(outpoint,inpoint)
      end do(iloop)
return
end
```

Figure 15. Fortran-like code for the transformation of the third index.

```
subroutine mp2(numk,numl,l_list,k_list,nblocks)

    comment: This routine calculates the second order Moller Plesset energy from
        transformed integrals distibuted on the basis of their third and fourth indices.
        numl and numk contain the number of k and l index values for the integral
        blocks located on the processor. k_list and l_list are arrays containing the lists
        of k and l index values. nblocks contains the total number of valid integral
        blocks in the column of processors.

    dimension k_list(50), l_list(50)

    comment: Loop over resident integral blocks, sending valid blocks down the
        processor column.

    do lcount = 1,numl
        l=l_list(lcount)
        do kcount = 1,numk
            k=k_list(kcount)

            comment: Skip this block if l is not an occupied orbital or k is not an
                unoccupied orbital.

            if (l.gt.nocc) goto 200
            if (k.le.nocc) goto 100

            comment: Send valid blocks to lower neighbor.

            call sendblk(k,l)

            comment: Receive a block from upper neighbor

        50  call recvblk(owner,visiting_k,visiting_l)
            nblocks=nblocks-1

            comment: If received block is a resident block, make
                contributions, but don't pass it on.

            if (owner.eq.mynode) then
                call contrb(visiting_k,visiting_l,value)

                comment: If all blocks in column have been received, then quit.

                if (nblocks.eq.0) goto 500

                comment: Otherwise send out another resident block.

                goto 100

            end if
```

Figure 16. FORTRAN-like driver routine for type 1 second order Moller Plesset energy calculation

```
      comment: Send visiting block to lower neighbor.
      call sendblkon
      comment: Calculate E(2) contribution from visiting integral block.
      call contrb(visiting_k,visiting_l,value)
      comment: Receive next block.
      goto 50
  100 continue
      comment: Now all valid resident integral blocks have been
               processed, but it is still necessary to wait for
               contributions from other processors' blocks.
  200 continue
      call recvblk(owner,visiting_k,visiting_l)
      call sendblkon
      call contrb(visiting_k,visiting_l,value)
      comment: Quit if last block has been processed.
      nblocks=nblocks-1
      if (nblocks.eq.0) goto 500
      goto200
      comment: All that remains is to sum up the partial E(2) values on the host.
  500 call hostsum(value)
      return
      end
```

Figure 16. continued

Total Execution Time (secs)			
# Nodes	Integral	SCF	Fock Matrix
3	811.94	1289.20	7.86
7	352.06	706.21	3.14
15	158.66	479.92	1.49
21	79.14	378.62	0.70

Speedups				
# Nodes	Integral	SCF	Fock Matrix	Ideal
3	1.00	1.00	1.00	1.00
7	2.34	1.83	2.25	2.33
15	4.92	2.69	5.05	5.00
31	10.0	3.40	10.31	10.33

Load Imbalance (%)			
# Nodes	Integral	SCF	Fock Matrix
3	1.4	--	11.9
7	5.9	--	4.9
15	9.1	--	17.3
21	11.3	--	14.6

Figure 17. SCF Benchmark results for 24 basis function C_2H_2

Total Execution Time (secs)			
# Nodes	Integral	SCF	Fock Matrix
1	364.83	5167.36	2.05
3	116.42	2200.32	0.67
7	50.93	141.74	0.34
15	25.97	113.94	0.26
31	11.78	101.58	0.16

Speedups				
# Nodes	Integral	SCF	Fock Matrix	Ideal
1	1.00	1.00	1.00	1.00
3	3.13	2.35	3.05	3.00
7	7.16	3.65	6.10	7.00
15	14.05	4.535	8.00	15.00
31	30.98	5.09	12.80	31.00

Load Imbalance (%)			
# Nodes	Integral	SCF	Fock Matrix
1	--	--	--
3	4.5	--	7.1
7	15.0	--	23.8
15	28.2	--	56.3
21	16.8	--	72.5

Figure 18. SCF Benchmark results for 13 basis function H_2O

Total Execution Time (secs)		
# Nodes	Total SCF	Integral and Fock Matrix
3	19623.50	838.99
7	8573.07	358.88
15	4232.83	170.42
31	2248.43	84.02

Speedups			
# Nodes	Total SCF	Integral and Fock Matrix	Ideal
3	1.00	1.00	1.00
7	2.29	2.34	2.33
15	4.64	4.92	5.00
31	8.73	9.99	10.33

Load Imbalance (%)		
# Nodes	Total SCF	Integral and Fock Matrix
3	--	1.4
7	--	8.0
15	--	9.0
31	--	11.1

Figure 19. Direct SCF Benchmark results for 24 basis function C_2H_2

Total Execution Time (secs)		
# Nodes	Total SCF	Integral and Fock Matrix
1	11555.23	368.90
3	3749.52	117.82
7	1691.25	51.52
15	909.34	26.27
31	479.1	12.37

Speedups			
# Nodes	Total SCF	Integral and Fock Matrix	Ideal
1	1.00	1.00	1.00
3	3.08	3.13	3.00
7	6.83	7.16	7.00
15	12.71	14.14	15.00
31	24.12	29.83	31.00

Load Imbalance (%)		
# Nodes	Total SCF	Integral and Fock Matrix
1	--	--
3	--	2.9
7	--	14.9
15	--	28.1
21	--	19.9

Figure 20. Direct SCF Benchmark results for 13 basis function H_2O

Total Execution Time (secs)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
8	1213.02	863.42	9.28	341.84	4.59
16	649.12	804.93	4.85	163.70	2.66
32	399.15	730.43	2.58	81.26	2.19

Speedups						
# Nodes	Integral	SCF	Fock Matrix	Trans.	MPPT	Ideal
8	1.00	1.00	1.00	1.00	1.00	1.00
16	1.87	1.07	1.91	2.09	1.73	2.00
32	3.04	1.18	3.60	4.21	2.09	4.00

Load Imbalance (%)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
8	14.8	20.8	15.2	--	--
16	27.3	18.2	21.8	--	--
32	47.6	25.9	34.8	--	--

Figure 21. Type 1 Moller Plesset benchmark results for 24 basis function C_2H_2

Total Execution Time (secs)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
16	1242.32	945.88	5.09	385.60	0.48
32	764.34	925.42	3.09	190.58	0.24

Speedups						
# Nodes	Integral	SCF	Fock Matrix	Trans.	MPPT	Ideal
16	1.00	1.00	1.00	1.00	1.00	1.00
32	1.63	1.02	1.65	2.023	2.00	2.00

Load Imbalance (%)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
16	24.4	35.8	43.4	--	--
32	47.2	34.9	63.7	--	--

Figure 22. Type 2 Moller Plesset benchmark results for 24 basis function C_2H_2

Total Execution Time (secs)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
2	698.03	325.66	3.55	76.96	3.76
4	341.22	266.11	1.78	38.69	1.92
8	190.99	257.78	0.96	21.41	1.87
16	109.66	227.94	0.56	12.18	1.14
32	60.00	209.66	0.32	6.26	0.88

Speedups						
# Nodes	Integral	SCF	Fock Matrix	Trans.	MPPT	Ideal
2	1.00	1.00	1.00	1.00	1.00	1.00
4	2.05	1.22	2.00	1.94	1.96	2.00
8	3.66	1.26	3.70	3.60	2.01	4.00
16	6.37	1.43	6.34	6.23	3.31	8.00
32	11.63	1.55	11.10	12.30	4.27	8.00

Load Imbalance (%)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
2	9.6	20.6	12.6	--	--
4	18.3	23.5	21.6	--	--
8	39.0	28.9	33.3	--	--
16	53.8	25.9	45.7	--	--
32	61.3	18.0	65.0	--	--

Figure 23. Type 1 Moller Plesset benchmark results for 13 basis function H_2O

Total Execution Time (secs)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
2	1292.69	372.26	3.30	178.38	0.45
4	631.36	317.33	1.66	87.04	0.24
8	353.66	299.22	0.99	48.27	0.16
16	202.94	274.90	0.58	27.25	0.11
32	111.01	237.38	0.40	13.98	0.06

Speedups						
# Nodes	Integral	SCF	Fock Matrix	Trans.	MPPT	Ideal
2	1.00	1.00	1.00	1.00	1.00	1.00
4	2.05	1.17	1.98	2.05	1.87	2.00
8	3.66	1.24	3.32	3.70	2.80	4.00
16	6.37	1.35	5.72	6.54	4.00	8.00
32	11.65	1.57	8.24	12.76	7.00	16.00

Load Imbalance (%)					
# Nodes	Integral	SCF	Fock Matrix	Transformation	MPPT
2	9.6	33.5	22.3	--	--
4	9.4	36.6	16.3	--	--
8	39.1	39.9	58.1	--	--
16	51.5	38.0	69.4	--	--
32	62.7	29.3	88.0	--	--

Figure 24. Type 2 Moller Plesset benchmark results for 13 basis function H_2O

Processors		Basis Functions						
	Mesh	8	16	20	23	24	25	32
1	2x2	11.97	--	--	--	--	--	--
2	1x2	6.06	175.65	--	--	--	--	--
2	2x1	6.05	175.70	--	--	--	--	--
4	1x4	3.20	85.18	267.57	--	--	--	--
4	2x2	3.14	84.51	265.76	--	--	--	--
4	1x4	3.20	85.22	267.63	--	--	--	--
8	1x8	1.82	43.04	159.09	277.23	321.44	--	--
8	2x4	1.71	42.19	128.94	284.54	315.87	--	--
8	4x2	1.74	42.14	128.88	277.06	315.89	--	--
8	1x8	1.92	43.14	159.26	272.06	321.49	--	--
16	1x16	--	23.68	107.46	185.41	216.53	260.46	--
16	2x8	1.17	21.81	77.47	140.24	155.65	263.63	--
16	4x4	1.12	21.44	64.00	136.00	154.10	246.13	--
16	8x2	1.18	21.66	77.54	137.38	155.65	254.14	--
16	16x1	--	23.25	107.65	182.21	216.98	260.27	--
32	1x32	--	--	--	--	--	--	352.54
32	2x16	--	12.35	53.01	95.14	105.81	132.75	327.73
32	4x8	0.90	11.34	38.80	68.61	77.65	139.36	320.56
32	8x4	0.86	12.08	38.80	68.61	77.62	139.25	320.54
32	16x2	--	12.34	53.17	95.50	105.84	132.74	327.79
32	32x1	--	--	--	--	--	--	352.32

Figure 25. Summary of execution times in seconds for parallel transformation program.

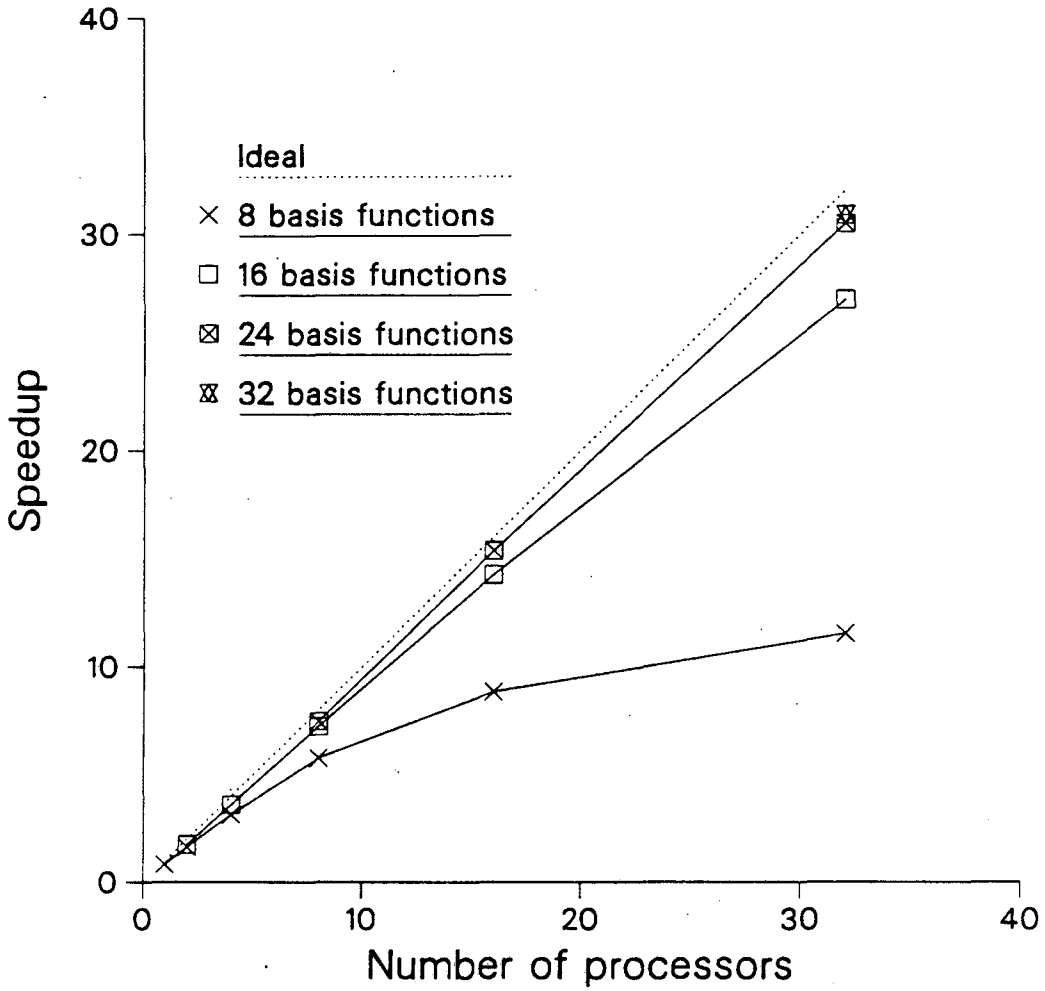


Figure 26. Speedup curves for test cases with even load balancing.

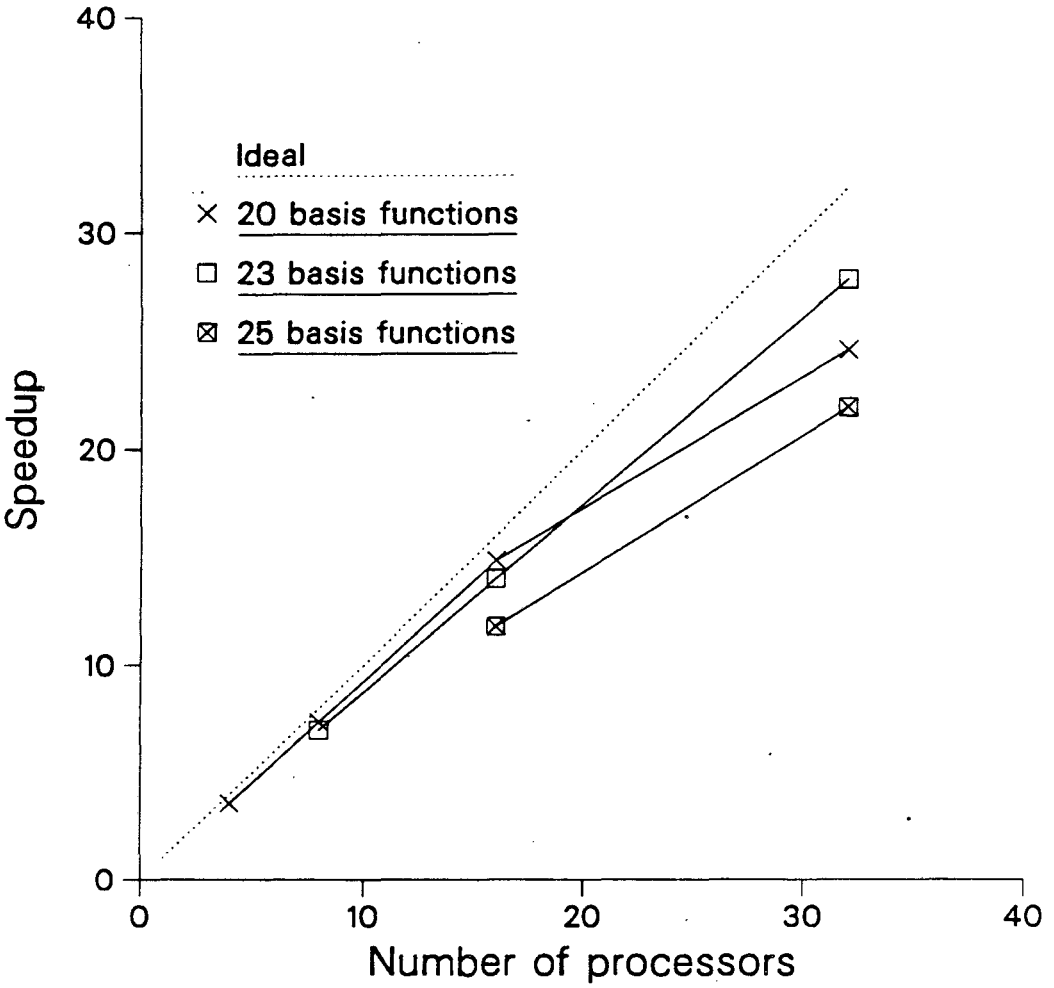


Figure 27. Speedup curves for problem sizes with unequal load distributions.

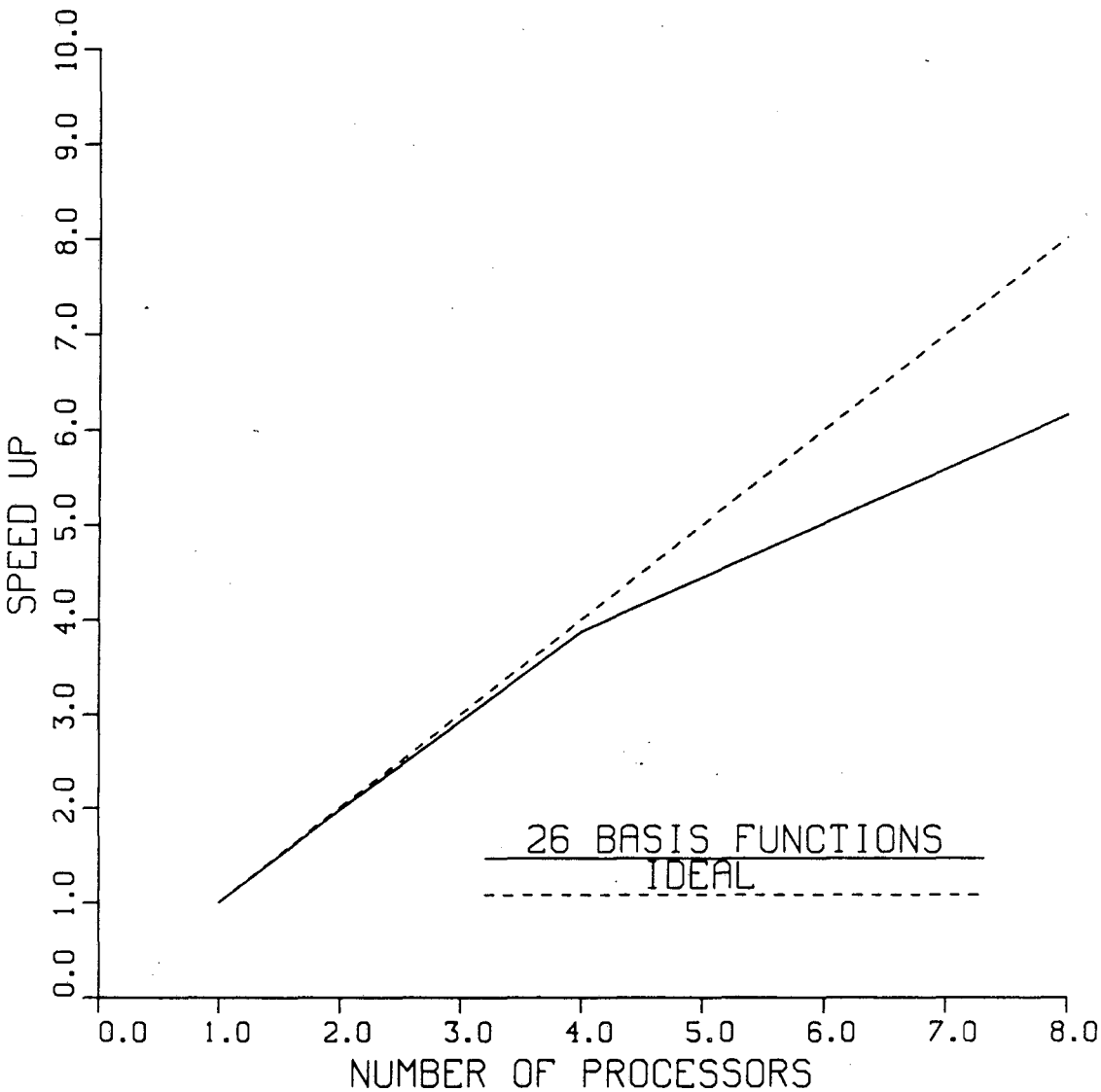


Figure 28. SCF speedup curves for a network of Sun Workstations.

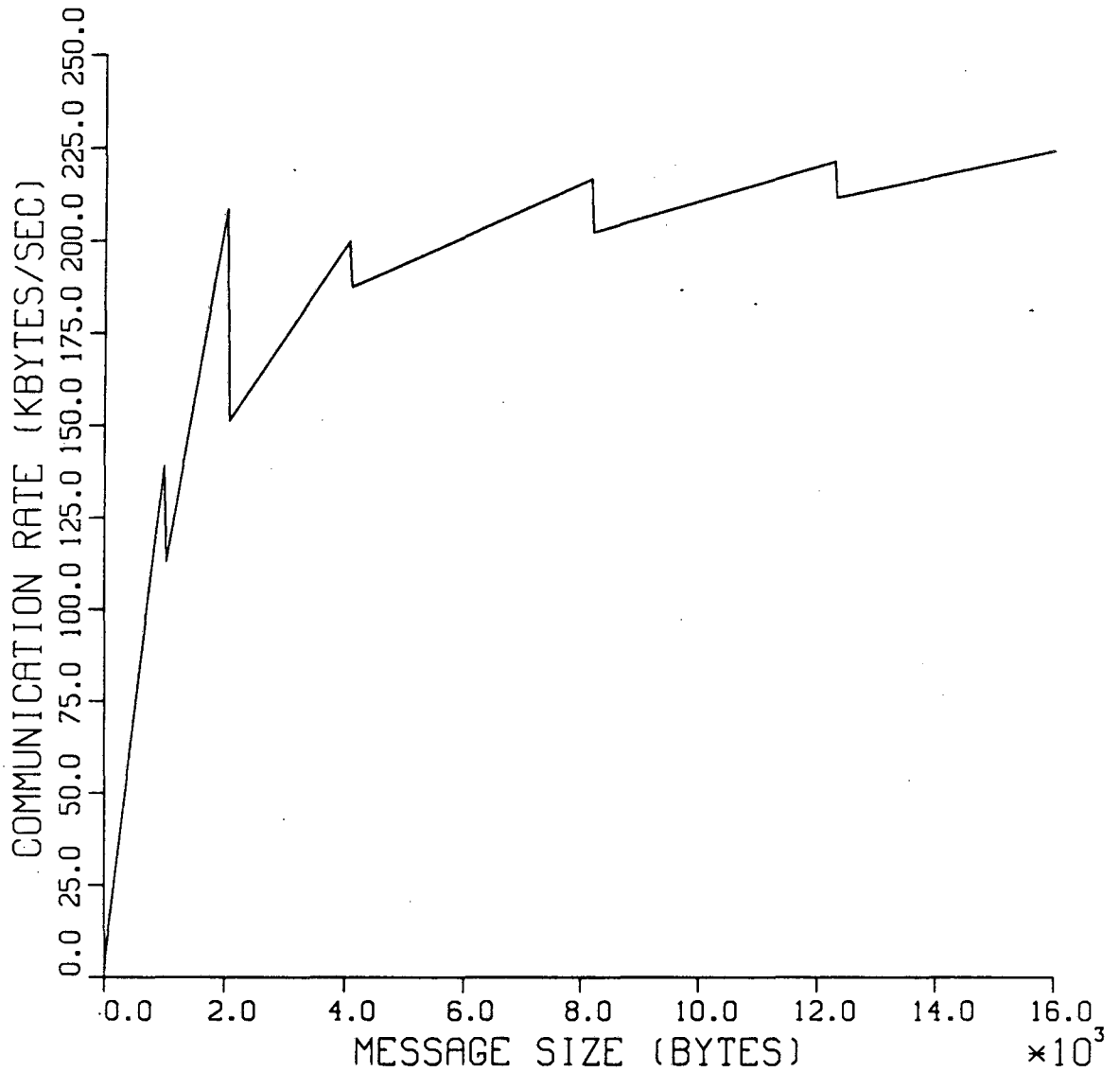


Figure 29. Node-to-node communication rate vs. message size.

References

[1]J. Von Neumann, transcript of talk given at the Institute for Advanced Study, May 15, 1946, Office of Naval Research, Washington D.C.

[2]N. Metropollis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, J. Chem. Phys. **21,** 1087 (1953).

[3]S.F. Boys, G.B. Cook, C.M. Reeves and I. Shavitt, Nature **178,** 1207 (1956).

[4]From the Cray-1 at 12.5 nsec. in 1976 to the Cray-2 at 4.1 nsec. in 1985.

[5]B.L. Buzbee, Tech. Report # LA-UR-83-1389, Los Alamos National Laboratories, (1983).

[6]D.L. Slotnick, W.C. Borck and R. McReynolds, Proc. Fall Jt. Computing Conf. AFIPS Conf. Proc. **22,** 97 (1962).

[7]G.H. Barnes, R.M. Brown, M. Kato, D.J.Kuck, D.J. Slotnick and R.A. Stokes, IEEE Trans. Computing **C-17,** 746 (1968).

[8]Many refs. in Y. Paker *Multi-Microprocessors* (Academic Press, New York, 1983).

[9]G. Fiellant and D. Rogers, Electronic Design, (Sept. 1984).

[10]FPS T Series, Sales Brochure from Floating Point Systems, Inc., Box 23489 Portland, Oregon 97223.

[11]U. Bernutat-Buchmann, D. Rudolph and K.H. Scholtter, *Parallel Computing Eine Bibliographie* (Univ. of Bochum, Bochum, 1983).

[12]N.S. Ostlund, R.A. Whiteside and P.G. Hibbard, J. Phys. Chem. **86,** 2190 (1982).

[13]E. Clementi, G. Corongiu, J.H. Detrich, L. Domingo, A. Laaksonen, H.L. Nguyen and S.Chin, Tech. Report # POK-40, IBM, (1984).

[14]S. Otto, *Caltech/JPL Concurrent Computing Project Annual Report 1983-84, vol. 2 Applications,* (Caltech, 1985).

[15]E.A. Hylleraas, Z. Physik **48,** 469 (1928).

[16]H.M. James and A.S. Coolidge, J. Chem. Phys. **1,** 825 (1933).

[17]J.H. Van Vleck and A. Sherman, Rev. Mod. Phys. **7,** 167 (1935).

[18]R.S. Mulliken and C.C.J. Roothan, Proc. Nat. Acad. Sci. **45,** 395 (1959).

[19]Many examples in H.F. Schaefer III, Science **231,** 1100 (1986).

[20]E. Clementi, G. Corongiu, J. H. Detrich, S. Chin and L. Domingo, Int. J. Q. Chem. Symposium **18,** 601 (1984).

[21]M. Dupuis, private communication.

[22]For examples where relativistic effects are significant: P.A. Christiansen, W.C. Ermler and K.S. Pitzer, Ann. Rev. Phys. Chem. **36,** 407 (1985).

[23]B.T. Sutcliffe, in *Computational Techniques in Quantum Chemistry,* edited by G.H.F. Diercksen, B.T. Sutcliffe and A. Viellard (Reidel, Boston, 1975), p 1.

[24]A. Szabo and N.S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory (Macmillan, New York, 1982).*

[25]I. Shavitt, in *Methods of Electronic Structure Theory,* edited by H.F. Schaefer III (Plenum, New York, 1977), p 189.

[26]J.A. Pople, J.S. Binkley and R. Seeger, Int. J. Q. Chem. Symposium **10,** 1 (1976).

[27]R.J. Bartlett, Ann. Rev. Phys. Chem. **32,** 359 (1981).

[28]C.F. Bender, J. Comp. Phys. **9,** 547 (1972).

[29]Y. Osamura, Y. Yamaguchi, P. Saxe, M.A. Vincent, J.F. Gaw, H.F. Schaefer III, Chem. Phys. **72,** 131 (1982).

[30]J.E. Rice, R.D. Amos, N.C. Handy, T.J.Lee and H.F. Schaefer III, J. Chem. Phys. **85,** 963 (1986).

[31]J.A. Pople, R. Krishnan, H.B. Schlegal and J.S. Binkley, Int. J. Q.

Chem. Symposium **13,** 225 (1979).

[32]Responses to First Survey of Parallel Processing Projects, compiled by Arvind, Laboratoty for Computer Science, MIT (1985).

[33]W.D. Hillis, *The Connection Machine* (MIT, Cambridge, 1985).

[34]D.J. Kuck, E.S. Davidson, D.H. Lawrie and A.H. Sameh, Science **231,** 967 (1986).

[35]J.E. Brandenburg and D.S. Scott, Tech. Report # 280182-001, Intel Scientific Computers, (1983).

[36]E. Felton, S. Karlin and S. Otto, Caltech/JPL Concurrent Computing Project Memo 92B.

[37]M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP Completeness* (W.H. Freeman, San Francisco, 1979).

[38]Review listing many refs: D. Hegarty and G. Van der Velde, Int. J. Q. Chem. **23,** 1135 (1983).

[39]S. Obara and A. Saika, J. Chem. Phys. **84,** 3963 (1986).

[40]H. Taketa, S. Huzinaga, K. O-ohata, J. Phys. Soc. of Japan **21,** 2313 (1966).

[41]B.N. Parlett, *The Symmetric Eignevalue Problem* (Prentice-Hall, Englewood Cliffs, 1980).

[42]C.G.J. Jacobi, J. Reine Angew. Math, **30,** 51 (1846).

[43]A.H. Sameh, Math. of Comp. **25,** 579 (1971).

[44]R.A. Whiteside, N.S. Ostlund, and P.G. Hibbard, IEEE Trans. Computing **C-33,** 409 (1984).

[45]R.P. Brent and F.T. Luk, SIAM J. Sci. Stat. Comp. **6,** 69 (1985).

[46]V.R. Saunders and J.H. van Lenthe, Mol. Phys. **48,**923 (1983).

[47]H.B. Schlegel, J. Chem. Phys. **84,**4530 (1986)

[48]E.R. Davidson, J. Comp. Phys. **17,** 87 (1975).

[49]C. Mead and L. Conway, *Introduction to VLSI Systems* (Addison-Wesley, Reading, 1980), pp 264-270.

[50]P.J. Reynolds, D.M. Ceperley, B.J. Alder and W.A. Lester, J. Chem. Phys. **77,** 5593 (1982).

[51]R.A. Friesner, Chem. Phys. Lett. **116,** 39 (1985).

[52]R.A. Friesner, J. Chem. Phys. **85,** 1462 (1986).

[53]S. Eigenback and C. Sadler, BYTE (Aug. 1985), p181.

[54]T. Toffoli, Physica **10D,** 117 (1984).

[55]U. Frisch, B. Hasslacher and Y. Pomeau, Phys. Rev. Lett. **56,** 1505 (1986).

[56]Tech. Report # 86.14, Thinking Machines Inc., Cambridge (1986).

[57]S. Wolfram, J. Stat. Phys. **45,** 471 (1986).

[58]P. Clavin, D. d'Humieres, P. Lallemand and Y. Pomeau, Compt. Rend. Acad. Paris, in press, (1986).

[59]D. Montgomery and G.D. Doolen, Tech. Report # LA-UR-86-2975, Los Alamos National Laboratories, (1986).

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720