JMENTATION PAGE

Form Approved
OMB No. 0704-0188

AD-A224 153

2. REPORT DATE: July 1990

3. REPORT TYPE AND DATES COVERED: professional paper

4. TITLE AND SUBTITLE

SUPERCONCURRENT PROCESSING—A DYNAMIC APPROACH TO HETEROGENEOUS PARALLELISM

5. FUNDING NUMBERS

PE: 0602234N
WU: DN300086
PR: ECB2

6. AUTHOR(S)

R. F. Freund

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Ocean Systems Center
San Diego, CA 92152–5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Ocean Systems Center
Block Programs
San Diego, CA 92152–5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper describes an approach to finding and using an optimal, heterogeneous suite of processors to solve supercomputing problems. This technique, called superconcurrency, currently works best when the computational requirements are diverse and significant portions of the code are not tightly-coupled. It is also dependent on new methods of benchmarking and code profiling, as well as eventual use of AI techniques for intelligent management of the selected superconcurrent suite. This latter technique, combined with anticipated bandwidth increases, will permit much more closely coupled code portions to be distributed on the heterogeneous suite. This paper also presents theoretical and empirical results to show SIMD architectures are faster than vector architectures for processing long vectors. Implications for future architectures and distributed heterogeneous processing in general are also discussed.

Published in *IEEE Proceedings*, February 1990.

14. SUBJECT TERMS

massive parallel processing,
high performance computing

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS ABSTRACT

SUPERCONCURRENT PROCESSING - A DYNAMIC APPROACH TO HETEROGENEOUS PARALLELISM

Richard F. Freund, Naval Ocean Systems Center

ABSTRACT. This paper describes an approach to finding and using an optimal, hetereogeneous suite of processors to solve supercomputing problems. This technique, called superconcurrency, currently works best when the computational requirements are diverse and significant portions of the code are not tightly-coupled. It is also dependent on new methods of benchmarking and code profiling, as well as eventual use of AI techniques for intelligent management of the selected superconcurrent suite. This latter technique, combined with anticipated bandwidth increases, will permit much more closely coupled code portions to be distributed on the heterogeneous suite. This paper also presents theoretical and empirical results to show SIMD architectures are faster than vector architectures for processing long vectors. Implications for future architectures and distributed heterogeneous processing in general are also discussed.

KEYWORDS. SIMD, vector processing, distributed processing, heterogeneous processing, superconcurrency, vecops, supercomputing, code profiling, benchmarking, optimal selection, Amdahl's Law

1. BACKGROUND

1.1. INTRODUCTION There is a growing consensus [1] among supercomputer scientists that super-speed computers of the future will be parallel processors, since the traditional vector processors are only able to pipeline out one or a few results per cycle. Parallel processors are potentially able to have hundreds or thousands of execution streams going at once. In earlier years, the clock cycle speed of supercomputers was so much faster than parallel processors and the parallel machines were in such an experimental state that it still made sense to look to vector processing for practical supercomputing. Both of those conditions are changed today, providing computational scientists with the occasion to begin to realize the full power of parallel processing. As we start to do this, however, we realize that while vector processors are basically all very similar to each other, parallel architectures present a wide variety of types. It seems quite unlikely that one of these types will be ideal for a wide range of problems. For that reason, a number of computational scientists are looking at **distributed heterogeneous processing** as a potential solution. Superconcurrency is one approach to this form of computing.

1.2. VECTOR ARCHITECTURES Vector architectures, such as the CRAY XMP (which will be the primary example in this paper), are primarily means for the hardware to support pipelining (Freund [2]). Suppose we wish to add a set of $\{x_i\}$ to a set of $\{y_i\}$, i.e., $\{z_i\}=\{x_i\}+\{y_i\}$. We refer to Fig 1.1 to see how this is normally done on a vector machine. The $\{x_i\}$ are loaded into one vector register (called V_0 here) and the $\{y_i\}$ into another (V_1) vector register. These operands are then fed through the floating point add unit and the $\{z_i\}$ are then pipelined out at the rate of one per clock cycle.

Most vector architectures are able to get some additional concurrency or parallelism by having some of the features of (a) two or three functional units in the pipeline stream (called chaining), (b) independent execution of the scalar portion of the processor, and (c) several copies of the scalar/vector CPU. Still the potential concurrency available in vector machines is quite limited and unlikely ever to have hundreds, much less thousands of execution streams going at once.

There are several idosyncrasies characteristic of vector machines. For example, methods of organizing memory are such that often stepping through memory (called stride) in units greater than one (as defined by the reverse lexicographic order implicit in FORTRAN) can result in significant performance degradation through memory conflicts. The author demonstrated several years ago a typical result, Fig 1.2., in which it is clear that the greater the power of 2 in the stride, the worse the performance (due to bank and section conflicts).

Another idiosyncrasy concerns linear algebra. Let us examine chaining in a basic linear algebra operation, matrix times a vector. Let $\mathbf{X}^T = (X(1),X(2),\ldots,X(N))$ be a point in N-space and $\mathbf{A} = (A(I,J))$ be the M by N matrix mapping $\mathbf{X}$ into M-space, i.e., $\mathbf{AX} = \mathbf{Y} = (Y(1),Y(2),\ldots,Y(M))^T$

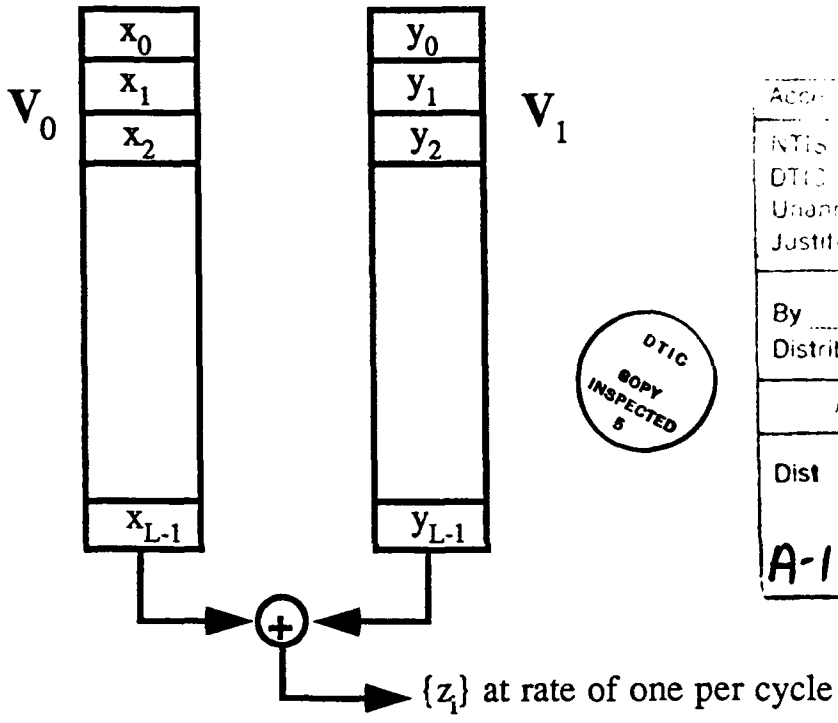

Fig 1.1 *Vector Pipelining of* $z_i = x_i + y_i$

Fig 1.2 *Memory Contention Effects on Pipelining of* $z_i = r^*x_i + y_i$

$$\begin{pmatrix} A(1,1), A(1,2), \ldots, A(1,N) \\ A(2,1), A(2,2), \ldots, A(2,N) \\ \\ \\ A(M,1), A(M,2), \ldots, A(M,N) \end{pmatrix} \begin{pmatrix} X(1) \\ X(2) \\ \\ \\ X(N) \end{pmatrix} = \begin{pmatrix} Y(1) \\ Y(2) \\ \\ \\ Y(M) \end{pmatrix}$$

where $Y(I) = \sum_J A(I,J) * X(J)$.

Algorithmically we can think of this in two ways: (a) as N updates to the elements of Y by successively adding in terms A(I,J)*X(J), commonly called SAXPY, or (b) as M dot products of rows of A with the column vector of X aka SDOT. Both are written in FORTRAN below (assuming initialization of Y to 0):

```
SAXPY                DO 1 J=1,N
                     DO 1 I=1,M
                1    Y(I) = Y(I) + A(I,J)*X(J)

SDOT                 DO 1 I=1,M
                     DO 1 J=1,N
                1    Y(I) = Y(I) + A(I,J)*X(J)
```

While the SDOT method corresponds to the way we have normally been taught to think theoretically of linear algebra operations, SAXPY is the method that works better on most vector architectures because of the nature of the hardware (essentially, in this case, the inability to add a vector to a scalar).

One of the consequences of these idiosyncrasies is the way people think about performing and benchmarking code for super-speed architectures. Most of the standard analysis tools, e.g., LINPACK, [3] use code strongly configured to implement SAXPY and avoid non-unit FORTRAN stride. However these rules of thumb, learned from vector architectures do not necessarily apply to parallel processors. NOSC's Superconcurrency Research Team (SRT) has striking examples where natural, parallel implementation of fundamental algorithms yields dramatic performance increases over traditional vector implementations (and associated limitations).

1.3. TYPES OF PARALLELISM One of the fundamental facts of parallel processing is the wide variety of types. There are a number of variant factors, e.g., memory organization (distributed, global, hierarchical, etc.) or processor interconnect scheme (bus, mesh, hypercube, etc.). However the most basic distinction is whether the processors execute the same instruction on multiple data (SIMD) or multiple instructions on multiple data (MIMD). Fig 1.3. below summarizes these types of paralellism compared to vector processing, with asymptotic performance factors. Since the time to execute on each MIMD processor often cannot be determined until run-time, there is some probability that many processors may have to wait for one to finish. Naturally this probability tends to increase as the number of processors increase, so that MIMD machines usually do not have thousands of processors. On the other hand each processor in a SIMD machine is usually a simple, e.g., bit-slice, processor (sometimes with an associated coprocessor) so that the execution time for any one processor is long. Thus SIMD machines do well only when the number of different data streams is quite large, i.e., in the thousands. The variety of parallel processors is also increased by such features as very long instruction word (VLIW) design, data-flow technology, and the fact that many designs are hybrids incorporating several different features. The fundamental result is that most parallel architectures are a good fit for some problems and a poor fit for others. The consequence is that an optimal method (to be made more precise in section 3. below) to compute a wide diversity of computational types is with a corresponding variety of architectures, i.e., the distributed heterogeneous processing approach mentioned in the introduction.

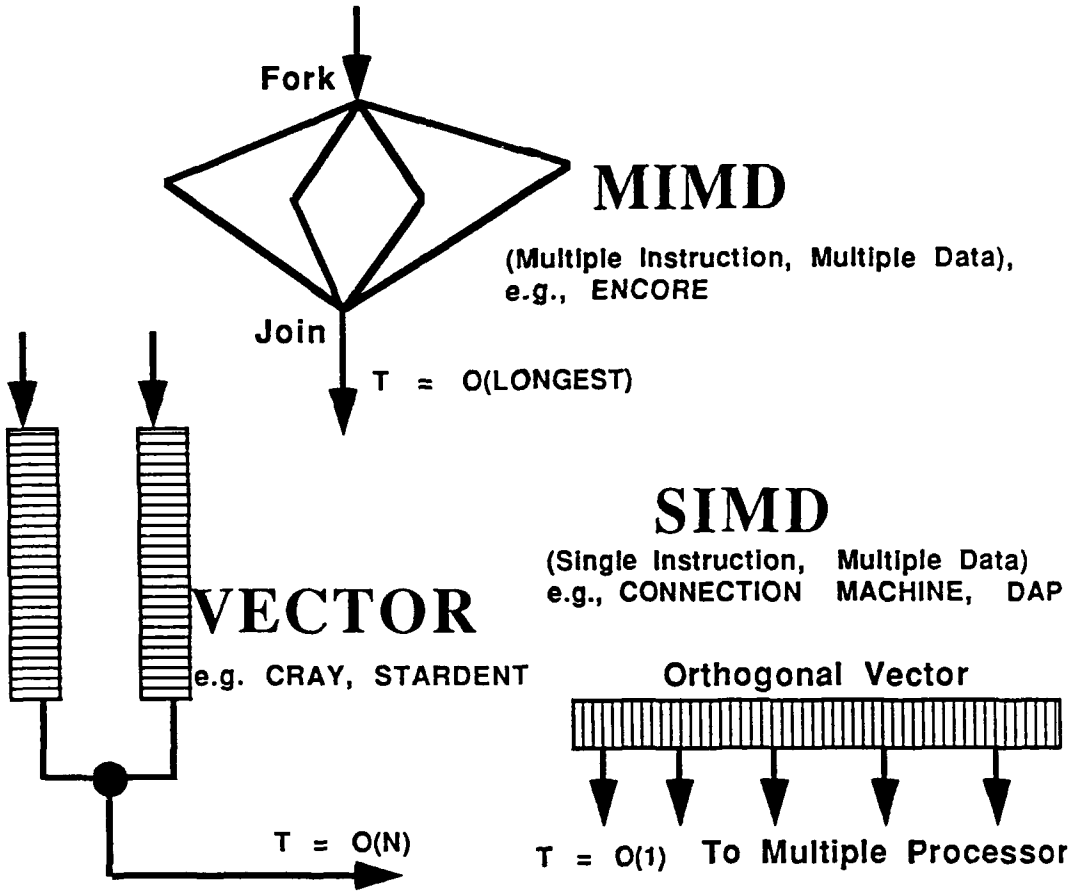

Fig 1.3 *Vector, SIMD, and MIMD Architectures*

2. SUPERCONCURRENCY

2.1. DEFINITION

Superconcurrency is a general technique for matching and managing optimally configured suites of super-speed processors. In particular this paper shows a general method for choosing the most powerful suite of heterogeneous parallel and vector supercomputers for a given problem set, subject to a fixed constraint, such as cost. The dual problem could find a minimal cost configuration for a fixed speed requirement. Thus the Optimal Selection Theory is a mathematical program for which one wishes to minimize the total time spent on the sum of all code subsegments. The theory is mathematically dependent on a new methodology of code profiling and a new methodology of analytical benchmarking. The intent is to use this technique to provide supercomputing power for Naval Command and Control (C2) problems, however this paradigm should work for many classes of supercomputing problems. The basic result is that for a computational problem that has a diverse set of computational types, not all tightly-coupled, the optimal solution is a heterogeneous suite of parallel and vector processors rather than a single supercomputing architecture. This solution is called superconcurrency both because it is an approach to supercomputing and because it concurrently uses concurrent (vector and parallel) processors. Ercegovac [4] has recently looked at the feasibility of a suite of heterogeneous processors to solve supercomputing problems. Resnikoff [5] and Kamen [6] have examined the cost-effectiveness of supercomputers (one generally finds the smaller mini-supers to be more cost-effective than the largest-sized machines). Bokhari [7] has investigated partitioning problems among various types of processors. There are several reasons for partitioning. First many large codes have diverse computational types. Second, the various super-speed parallel and vector processors have quite different performance profiles on these types, often amounting to several orders of magnitude. It is a commonplace observation and a corollary of Amdahl's Law [8] that any single type of supercomputer, often spends most of its time computing code types for which it is poorly designed. If we could configure our processor suite so that

each processor could spend almost all its time on code for which it is well designed, the overall increase in speed could be orders of magnitude over what is now achieved by conventional supercomputing.

2.2. REASONS FOR SUPERCONCURRENCY One way of understanding the reasons for superconcurrency is to look at Amdahl's Law. Basically this says that the overall rate at which a machine will compute an overall code or set of codes is determined by the sum of the inverses of the times on each subportion. The paradoxical consequence of this is that, in the face of diverse computation requirements, a single machinc asked to excute all the code will spend **most of its time on the portions of code for which it is not well designed**, as illustrated in fig 2.1 below. The superconcurrency approach is also shown here in which we try to identify and use a suite of machines wherein each is used primarily to compute code types for which it is well-suited, and conversely each portion of code is matched to an appropriate architecture.

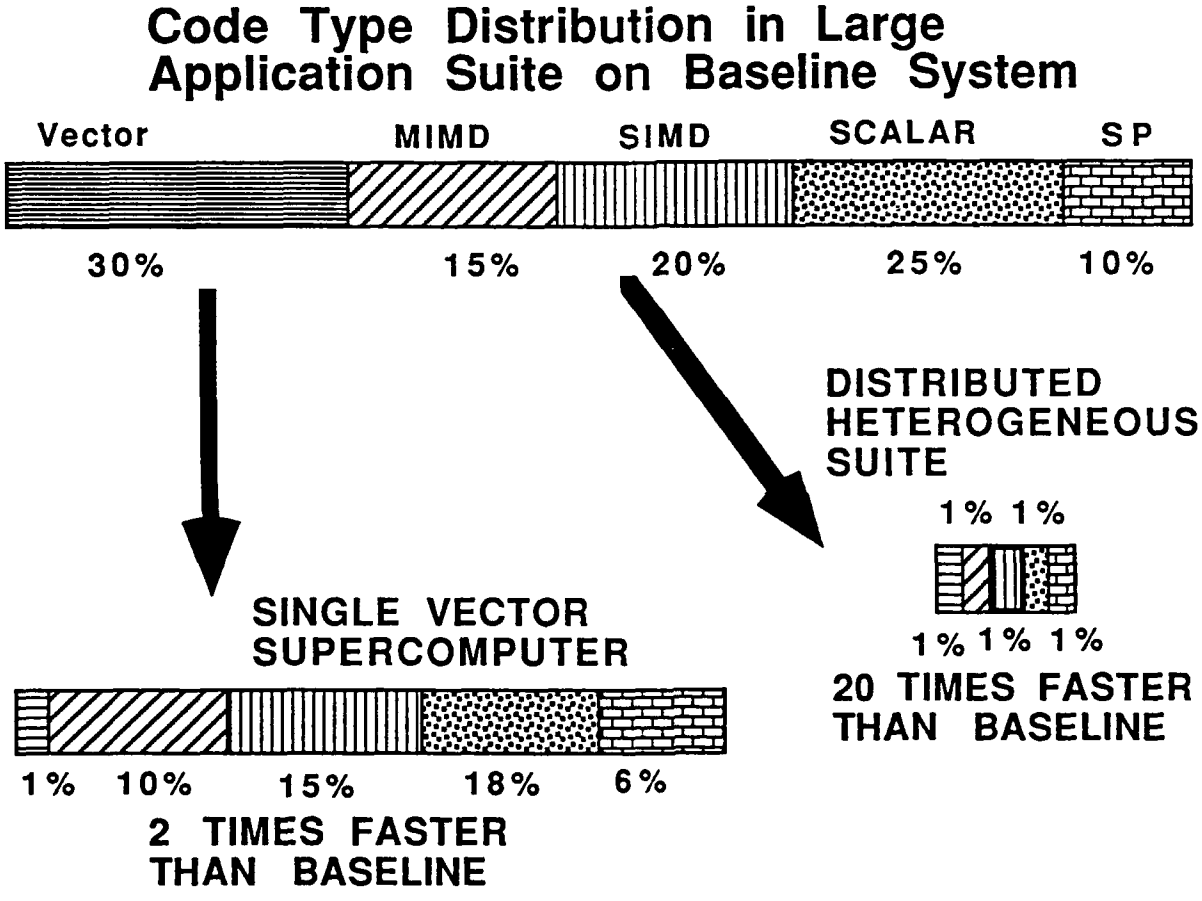

Fig. 2.1. *Code Profiling and Machine Matching*

2.3. BENCHMARKING AND CODE PROFILING As discussed earlier, the basic approach of this paper is contingent upon breaking down the overall code into groups of segments within which the processing requirements are the same or homogeneous. The segments of homogeneous type are assigned to optimal processors for that type. Before that can be done, it is necessary to take two benchmarking type steps. The first, called code-type profiling is a code specific function to identify the "natural" types of code that are actually present and group the code segments by type. Types that might be identified include vectorizable decomposable, vectorizable non-decomposable, fine/coarse-grain parallel, SIMD/MIMD parallel, scalar, special purpose, e.g., FFT or specialized sorting algorithm, etc. The second step, called analytical benchmarking is an analysis of how the available processors perform on the identified types, i.e., this identifies processors that are appropriate solutions for each code type. Thus it is more analytical than some previous techniques that simply looked at the overall result of running a processor on an entire benchmark code or set of loops (without any real

analysis of how the myriad of relevant factors contributed). However it should be pointed out that recent research by Dongarra on LINPACK provides some insight to the processes involved. Both code profiling and analytical benchmarking are now being undertaken by the Superconcurrency Research Team (SRT) at the Naval Ocean Systems Center (NOSC). Our initial research at Profiling/Benchmarking was directed at several large Naval C2 problems and a suite of potentially matching mini-supers/parallel processors (including the Connection Machine, DAP, Ardent, Encore, Butterfly, MultiFlow, Aspen, and Convex). Most of the C2 applications we have looked at so far have been relatively loosely-coupled and we have found it feasible to break them up (manually) into homogeneous portions and assign them to appropriate processors. From the processor (benchmarking) point of view, our most interesting result to date is how consistently the long vector problems are much better done on SIMD (Connection Machine or DAP) processors rather than vector processors.

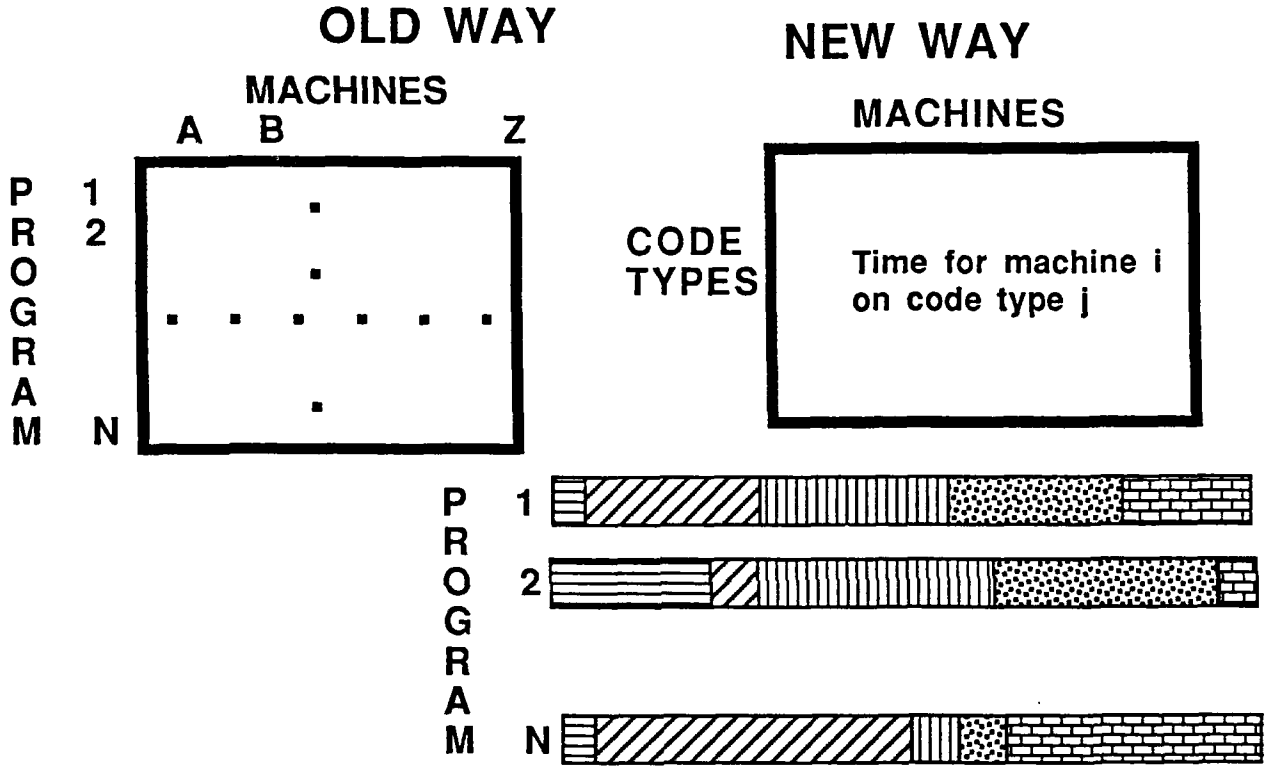

Fig. 2.2. *Analytical Benchmarking*

2.4. SIMD/VECTOR CROSSOVERS SIMD and vector architectures perform abstractly the same type of computation, since vectorization pipelines different data through the same functional unit. I call SIMD orthogonal vectorization (since the operations are done on a broad front, one deep, as opposed to a vector architecture which is N deep, but only one wide). Let us consider an elementary scientific calculation that has traditionally been done on vector machines, e.g., $\{z_i\} = \{x_i\} + \{y_i\}$, $i = 1, \ldots, N$. The x, y, and z variables are real numbers and N is typically some large integer in the hundreds or even thousands. Fig. 1.1 shows how this is normally done on a vector machine. The results are computed in time $O(N)$, or more precisely the time is bounded below by $N * \tau_v$, where τ_v is the clock cycle time of the particular vector machine in question.

A SIMD processor (Single Instruction Multiple Data), such as the Connection Machine or AMT DAP, typically has thousands of simple processors all executing the same instruction stream in lockstep. Figure 2.3. shows a method by which the same calculation could be computed on a SIMD architecture. Namely, $\forall$ i we load x_i , and y_i into processor i. Then we issue the same instruction, e.g., a floating point add, to all processors simultaneously. The add takes much longer on the simple SIMD processor than a comparable single add instruction on a vector machine. However, since all

processors are simultaneously computing the same instruction, the results are computed in time $\mathbf{O}(1)$, i.e, it takes the same time for any $N \leq M$, where M is the number of processors in the SIMD machine. Thus the time is bounded below by τ_s, where τ_s is the time needed for one of the SIMD processors to compute a floating point add. The implications of this are clear. If N is large enough such that $N * \tau_v > \tau_s$ then that total computation is performed faster on the SIMD than on the vector machine. The value of N for which the SIMD machine overtakes the vector machine, i.e., the least $N \ni N > \tau_s / \tau_v$ is called the **crossover point**, or **x-point** hereafter. Freund, Gherrity, and Kamen [9] computed x-points for several operations oriented around linear algebra computations (Lubeck [10]). One of these is $V = V + V$, e.g. $Z(I) = X(I) + Y(I)$, $I = 1,..., N$. The results of this computation are shown in Figure 2.4. We feel that the results of this experiment, run on a CRAY XMP, Convex 210, 4K & 8K Connection Machines (with floating point coprocessor), 1K & 4K DAP, and 4K DAP (simulated with coprocessors) support the conclusions that:

a. SIMD architectures are potentially faster than vector architectures for long vector problems.
b. The DAP appears to be a more efficient SIMD architecture than the Connection Machine

Until recently, vector problems of lengths in the thousands have not been usual. With increasingly more difficult problems of the future, e.g., in moving from 2D to 3D simulations, computational scientists may well need the long-vector capability of SIMD architectures.

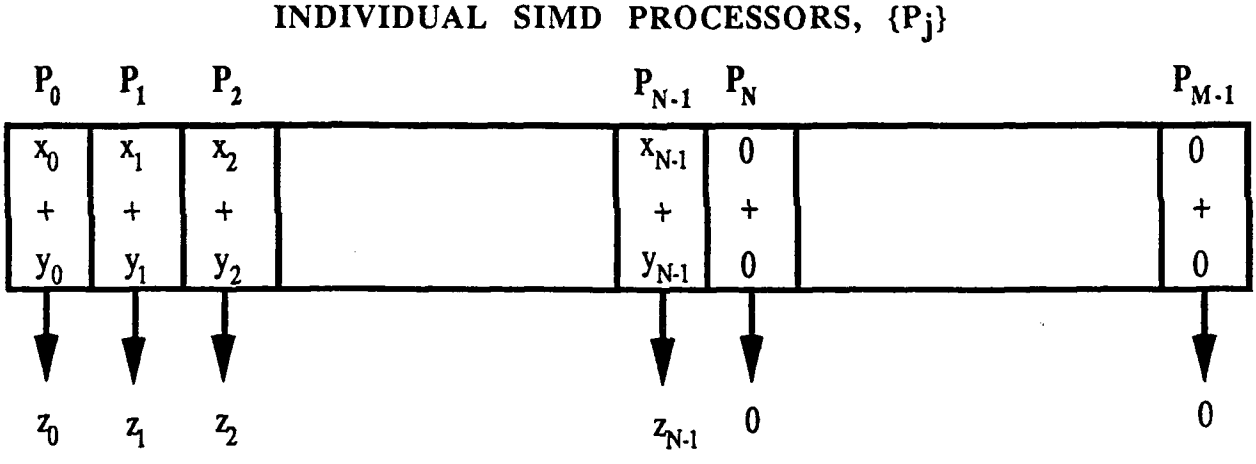

Fig 2.3. *SIMD, "ORTHOGONAL VECTORIZATION" OF* $z_i = x_i + y_i$

3. OPTIMAL SELECTION THEORY or STATIC OPTIMIZATION The Optimal Selection Theory is a mathematical program for which one wishes to minimize the total time spent on the sum of all code subsegments subject to a fixed cost constraint. The method is mathematically dependent on a new methodology of code profiling of the problem sets being implemented and a new methodology of analytical benchmarking. The full formulation of this theory is given by Freund [11].

3.1. MATHEMATICAL FORMULATION We can state the basic problem as a linear (actually integer) program. We want to get the most power we can, given some overall cost constraint. More mathematically we wish to maximize

the power (or speed) function, $\mathcal{P}$. We do this by minimizing a time function, $\mathcal{T}$, giving the time taken on a code, so

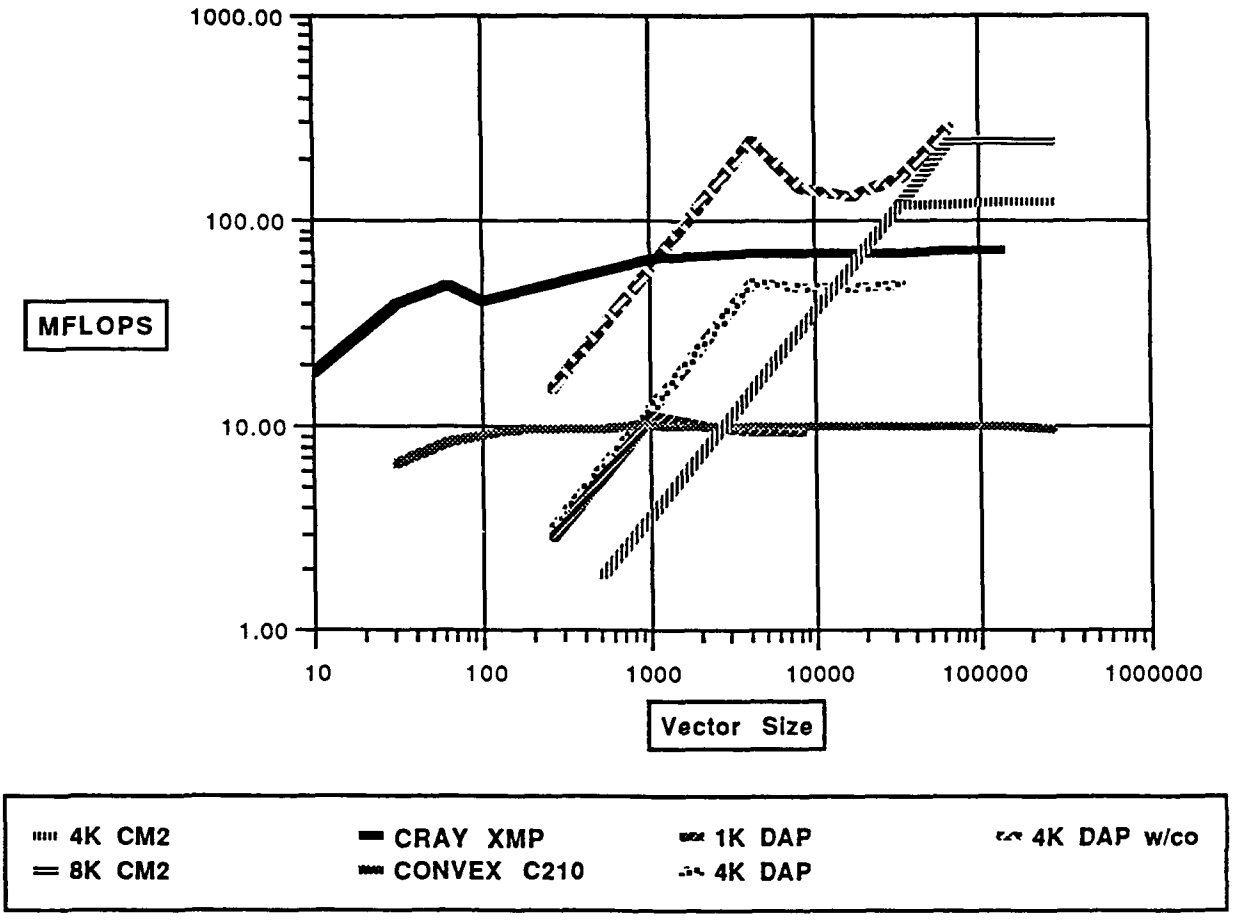

Fig 2.4. *SIMD/Vector Comparisons for* $z_i = x_i + y_i$

that $\mathcal{P} = \mathcal{T}^{-1}$. $\mathcal{T}$ is defined on the two-variable range, $\mathcal{X} \times \mathcal{S}$. $\mathcal{X}$ is the set of potential machine choices, $\mathcal{X} = \{x_i\}$ where the x_i are candidate architectures. $\mathcal{S}$ is a non-overlapping set of all code subsegments, s_j; thus $\mathcal{S} = \bigcup s_j$ and $s_j \cap s_k = \varnothing$ if $j \neq k$. The choice of the s_j defines the code profiling and analytical benchmarking problem. We denote $\mathcal{C}$ as the overall cost constraint, $\{c_i\}$ as the set of costs corresponding to the $\{x_i\}$ and $\{t_i\}$ as the set of corresponding time functions, i.e., $t_i(s_j)$ is the time taken by machine x_i on code segment s_j. Let $\mathcal{I}$ denote the set of all possible indices of one machine type per segment with v_i denoting the number of such machines used per segment. Let v_i be the number of machines of type i (which may be 0 if machine x_i not in the indexed configuration). Then the mathematical programming problem can be stated as:

$$(3.1) \qquad \text{MINIMIZE } \mathcal{T}(x_i, s_j) = \sum_{i \in I, j} \frac{t_i(s_j)}{\nu_i}$$

$$\text{such that } \sum_i \nu_i c_i \leq \mathcal{C}$$

3.2 EXAMPLE Let us consider the following example. Suppose the code to be 50% vectorizable (35% non-decomposable, i.e, only one vector machine at a time can run it, and 15% decomposable), 20% suitable for SIMD, 20% MIMD, and 10% inherently scalar. We shall assume that each type of machine only achieves scalar speed on code for which it is not designed, e.g., a vector machine will be assumed to get only scalar speed on parallel code. In Table 3.1. below we denote by α the speed up each machine achieves on portions of code for which it is best suited. The V's are vector machines, the S's SIMD, the M's MIMD, and the Sc a scalar machine. Suppose our overall cost constraint is \$4M.

	V_1	V_2	V_3	S_1	S_2	M_1	M_2	Sc
c (in \$M)	4	1	0.3	1	0.3	1	0.3	0.25
α	8	5	3	15	6	4	2	2

Table 3.1. *Optimal Selection Theory Example*

We can reformulate eq. 3.1 above as:

$$(3.2) \qquad \mathcal{T} = \sum_{j=1}^{N} \frac{p_j}{\nu_j} \sum_{i=1}^{M} \nu_i t_{i,j}$$

where N = # different code types, p_j = % of code type j, ν_i = total # processors for code type i. M = # processor types for code type j, and $t_{i,j}$ = time for processor i on code type j

In this computable form, we see the traditional vector supercomputer solution of 1 V_1 has $\mathcal{P} = 4.00$. However the multi-machine solution of 1 V_2, 3 V_3, 1 S_1, and 1 M_1, in which no one machine is a traditonal supercomputer, has a greater power function, $\mathcal{P} = 5.14$. This is true in spite of the fact that **50% of the code was assumed vectorizable.**

4. DINS OR DYNAMIC OPTIMIZATION Distributed Intelligent Network System (DINS) - One of the most active current research areas of the NOSC Superconcurrency Research Team (SRT) has been the development of the DINS concept. DINS will be a reasoning system that uses information from Code Profiling, Analytical Benchmarking, and network bandwidth to optimally manage a network of heterogeneous, high-performance, concurrent processors and assign portions of code to appropriate processors. In a general sense, this is similar to current research in load balancing and priority assignment. However the information to be used will be the three sources mentioned above with the primary aim of optimal matching code portions to processors rather than (the secondary) factors of load balancing and priority assignment. Since DINS will reason about processors actually available to it, this means we can achieve configuration control at different sites even though there may be a different superconcurrent suite at each. Similarly DINS will continue to function and assign a second best processor if a first choice is unavailable or down. Thus DINS is robust and survivable. Likewise it is compatible with evolutionary development; when a new processor is introduced because of

changing technology, we simply replace the old benchmarking data with the new. The features of robustness, configuration control, survivability, tailorability, and evolutionary development are essential for Naval C2 problems. We call DINS dynamic optimization since it dynamically tasks in an optimal way the backend suite of heterogeneous, superconcurrent processors that were chosen from the Optimal Selection Theory.

4.1. APPROACH We plan to use artificial intelligence and compiler writing techniques to build the DINS using an existing off-the-shelf high-level distributed operating system, e.g., CRONUS (BBN product) and MACH (DARPA-sponsored Carnegie Mellon product). We will then use the ongoing results of analytically benchmarking code profile types on a variety of machines for automating the partitioning of complex codes so that homogeneous portions can be sent to the best suited processors. Our supeconcurrency efforts will also draw on the developing taxonomy of code profile types with similar processing requirements, as well as our current work on the code profile types to find out what machines are ideal. Some code portions may be complex mixes of simple codes which are not easily decomposable because of, for example, unusual data dependencies in the algorithms.

4.2. EXAMPLE An example of how DINS would work can be seen from the SIMD/Vector crossover point study. DINS would have matrices of the x-points for the various vector and SIMD machines available on its network. A vector problem that was short would be done on a traditional vector machine; a long one on a SIMD machine. The kind of reasoning DINS would do would be similar in general nature to the reasoning involved in the now classical problem of load-balancing, but the data it would reason about would be the performance matrices determining optimal machine/code portion matching. Load balancing could, in fact, be a secondary consideration, but only secondary, since the performance increases one gets from this are typically much less than from superconcurrent matching.

4.3. EXPECTED RESULTS The findings of this project will enable us to assess the potential for improvements in performance from a heterogeneous mix of concurrent processors. Based on the findings of our Optimal Selection Theory, we expect that lower cost multi-machine solutions will have speedups better than what can be achieved with even the most powerful single super-computer. With an intelligent system to distribute tasks among multiple processors having disparate capabilities based on the code type, two to three orders of magnitude of speedup could be achieved. The intelligent system for distributing appropriate code should prevent problems of low vectorization fractions for the vector machines. We expect the various parallel and supercomputer machines to come closer to their peak performance ratings when they run code for which they are optimal. Another of the advantages of constructing a system which can access multiple processors as needed is that new computing technologies can be seamlessly incorporated into the system as they become available. The end users of the system need not learn any new interfaces to take advantage of improvements in technology. We can also expect fault tolerance from the ability to choose a second-best processor when one of the machines is unavailable, implying robustness. This reasoning about what is locally and currently available also implies automatic configuration control since DINS can run transparently at different sites with different back-end supercomputers. This also implies graceful evolutionary acquisition, as well as survivability and tailorability, all important consideration for Navy C2 environments.

5. FEASIBILITY An important issue in Superconcurrency is the feasibility of switching machines for various codes or subcodes in our applications suite. In this section we look at several aspects of this, and mention related research.

5.1. LEVELS Superconcurrency could be conducted at three distinct levels. The coarsest or highest level would be one in which we optimally match distinct whole codes to separate machines. The medium level granularity would correspond to sending different subroutines or largely autonomous subportions to optimal processors. The finest or lowest level would be the one at which we break up tightly-coupled portions of code in order to optimally match them to hardware. Clearly the coarsest level is easiest to implement, but yields the least performance, whereas the lowest level granularity is hardest, but gives the best results. Clearly a fundamental issue is the interprocessor bandwidths. Fortunately ranges exceeding 1Gbit and beyond should be readily achievable in the near future.

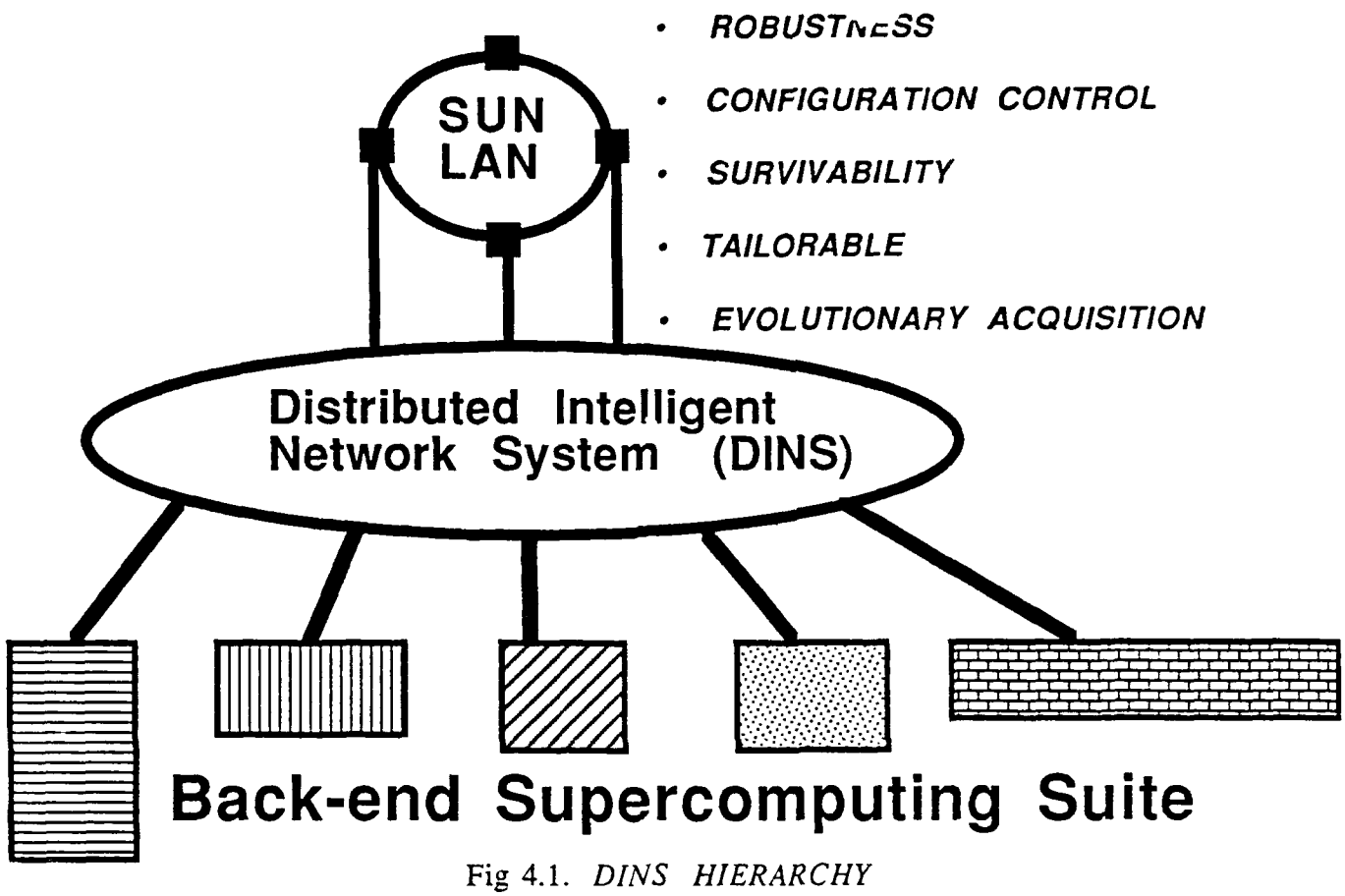

Fig 4.1. *DINS HIERARCHY*

5.2. BANDWIDTHS AND MIXED TYPES Tightly and medium-coupled portions of code will be more difficult to break up and assign to different processors and the ability to do this will rest in part on the bandwidths of the storage devices and distributed network used. In these cases, it may be necessary to assign mixed type code to the best processor available. Superconcurrent implementations will attempt to work at the lowest level compatible with the bandwidths available at any given site. Put another way, eq. 3.1 above will actually use $t'_{i,j}$ where the t' reflect not only the actual compute time for processor i on code type j, but the requried interprocessor communication time:

$$\mathcal{T} = \sum_{j=1}^{N} \frac{p_j}{v_j} \sum_{i=1}^{M} \nu_i \, t'_{i,j} \qquad (5.1)$$

5.3. CONCURRENT SUPERCOMPUTING Dr. Paul Messina [12] of JPL/CalTech will be implementing distributed heterogeneous processing using specialized computational resources at CalTech, JPL, Los Alamos National Laboratory, San Diego Supercomputer Center, and Argonne National Laboratoy. He should be able to achieve at least medium granularity of code distribution, since he will be operating with an 800Mb network.

5.4. PASM Fineberg, Casavant, and Siegel [13] of Purdue have constructed a special prototype machine, PASM, able to compute in both SIMD and MIMD mode. This enables them to study the performance of various algorithms on different architectural configuration. In addition, PASM is able to switch modes in a single cycle, so that study of mixed-mode computation is possible. In particular, this machine makes it possible to study superconcurrency issues at the lowest possible level, even matching modes to individual lines of code!

5.5. SOFTWARE and ALGORITHMS Methodologies for developing parallel algorithms and the associated software issues, are not addressed here. However these are key research areas at many laboratories. SRT's current efforts in this area, including the use of parallel ADA, will be available as superconcurrency is implemented for Navy C2 centers.

6. IMPLICATIONS

6.1. C2 OR RESOURCE MANAGMENT Superconcurrency is a technique now being tested to support Navy Command and Control (C2) problems. Command and Control is somewhat similar to resource managment in the civilian world. The aim of the C2 centers is to provide commanders and their staffs with tools to plan and allocate resources. Superconcurrency would fit into a generic center in the manner shown in Fig. 6.1. Different kinds of users, Operations, Intelligence, etc. would link into a C2 environment that would have available a variety of general purpose resources, e.g., file servers, general purpose computers, etc. Part of the C2 center would be DINS that would take compute-intensive work and optimally allocate it to the variety of back-end super-speed processors available at the given site.

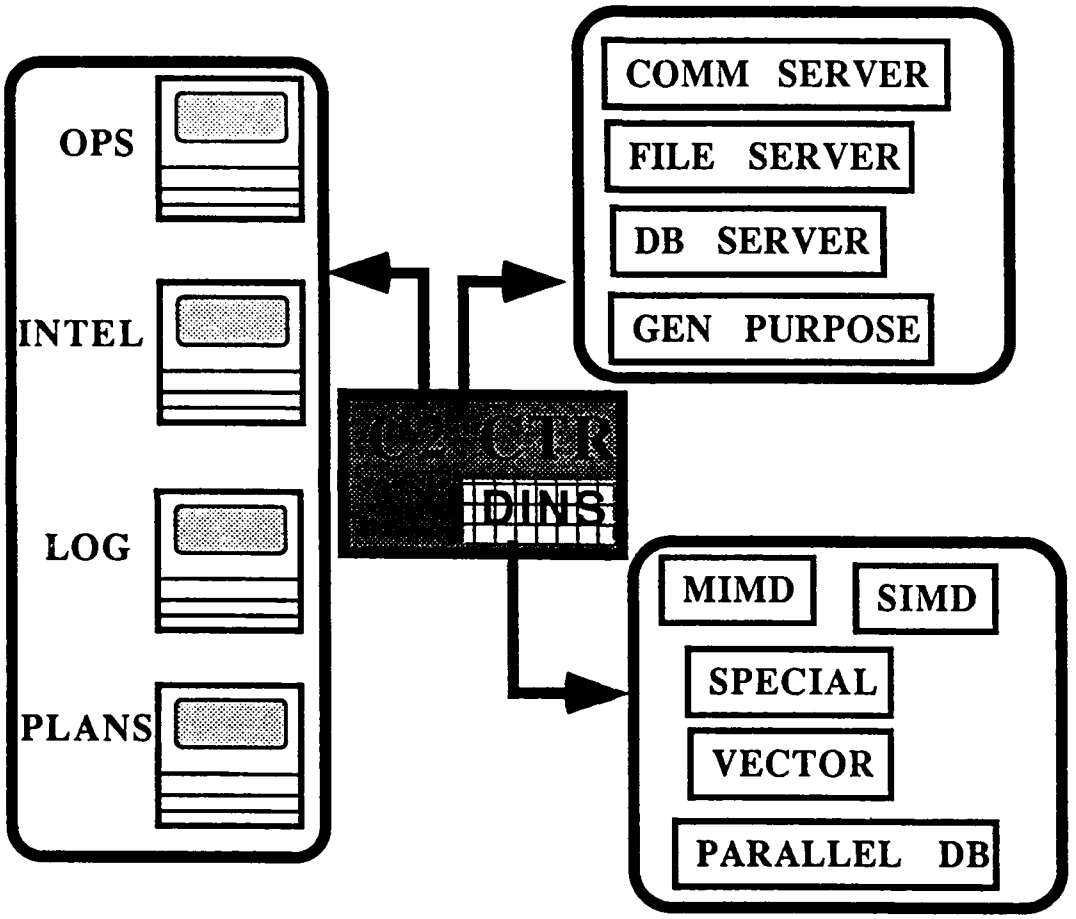

Fig 6.1. *DINS Role in Command Center*

6.2. SUPERCONCURRENCY POWER In order to be effective in a Navy C2 environment, Superconcurrency needs not just supply more computational power in the form of speed, but more generally it must supply a mix of speed, complexity (model fidelity), and multiplicity (what-ifs), as shown in Fig. 6.2. Furthermore, this mix must be easy to define at run-time by the user. The NOSC SRT has already demonstrated increases in speed of three orders of magnitude for some C2 models (by fitting them to the right processor type). The next step is to support, through DINS, the required power in the more general sense.

6.3. SUPERCONCURRENCY The underlying premise of this paper is that many codes, and particularly many sets of codes, have a heterogeneous set of computational types. The solution, called superconcurrency, is nothing more than the commonsensical approach of selecting a heterogeneous suite of processors that most effectively addresses this diverse set of requirements. The solution is expressed as a mathematical program with all that implies about the existence of an optimal solution. This approach requires a more analytical way of benchmarking and code profiling in order to analyze the power of various processors on atomic portions of code. Superconcurrency has the potential of achieving orders of magnitude greater speed over conventional supercomputers if the code profiling techniques show the overall application to be quite diverse in its requirements. The future addition of a Distributed Intelligent Network System to manage a superconcurrent suite of vector and parallel processors offers the potential of robustness, configuration control, survivability, tailorability, and evolutionary development.

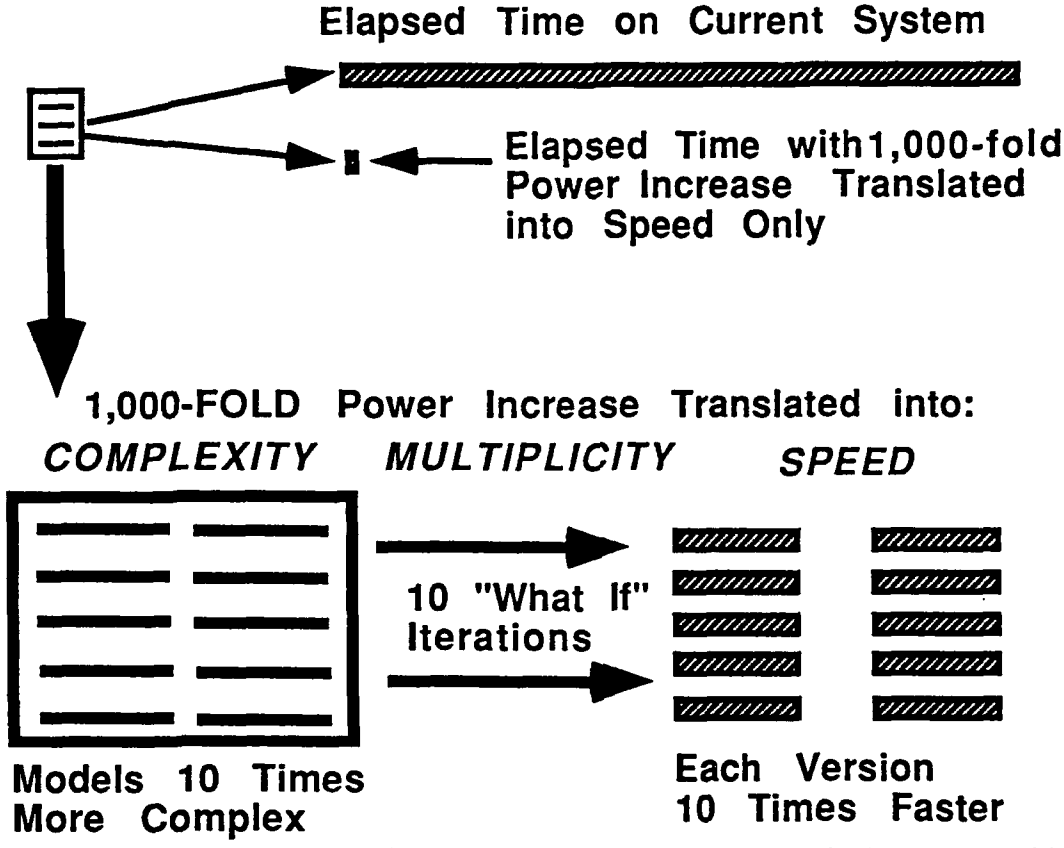

Fig 6.2. *Superconcurrency Power Applied to Baseline Model*

ACKNOWLEDGEMENTS

This research was supported by the Office of Naval Technology, under the Navy High Performance Computing Initiative, and the Naval Ocean Systems Center.

REFERENCES

[1] J. WORLTON, *Toward a Science of Parallel Computation*, Computational Mechanics, vol 75

[2] R. F. FREUND, *Linear Algebra on a CRAY XMP*, Private Communication

[3] J. J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*, Argonne National Laboratory Technical Memorandum No 23, Feb 12, 1989

[4] M. ERCEGOVAC, *Heterogeneity in Supercomputer Architectures*, Parallel Computing, vol 7 (1988), pp. 367-372.

[5] H. L. RESNIKOFF, *Cost-Effectiveness of Concurrent Supercomputers*, Journal of Supercomputing, vol 1 (1987), pp. 231-262.

[6] R. B. KAMEN, *Comparisons of Supercomputer Costs and Peak Performance*, Private Communication, May 1989
[7] S. H. BOKHARI, *Partitioning Problems in Parallel, Pipelined, and Distributed Computing*, IEEE Transactions on Computers, vol 37 (1988), pp. 48-57.
[8] G. M. AMDAHL, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capability*, Proc. AFIPS Comput. Conf., Vol 30, 1967
[9] R. F. FREUND, M. J. GHERRITY, R. B. KAMEN, *SIMD/Vector Crossover Points*, to be published
[10] O. M. LUBECK, *Supercomputer Performance: The Theory, Practice, and Results*, Adv. in Computers, 27, 309(1988)
[11] R. F. FREUND, *Optimal Selection Theory for Superconcurrency*, Proceedings of Supercomputing 89, Reno, NV (13-17 Nov 1989)
[12] P. MESSINA, *CalTech Concurrent Supercomputing*, Conference notes, January, 1990
[13] S. A. FINEBERG, T. L. CASAVANT, H. J. SIEGEL, *Experimental Evaluation of SIMD PE-Mask Generation and Hybrid Mode Parallel Computing on Multi-Microprocessor Systems*, Purdue University, TR-EE 88-55, July 1989