4

Technical Document 1669
October 1989

Optimal Selection Theory for Superconcurrency

R. F. Freund

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

E. G. SCHWEIZER, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This work was performed by the Ashore Command Centers Branch, Code 423, of the Naval Ocean Systems Center, San Diego, CA, with funding provided by the Chief of Naval Research, Code ONT-20P, Office of Naval Technology, Arlington, VA 22217-5000, under program element OMN.

Released by
R. E. Pierson, Head
Ashore Command Centers Branch

Under authority of
J. A. Salzmann, Jr., Head
Ashore Command and
Intelligence Centers
Division

FS

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1 AGENCY USE ONLY *(Leave blank)*	2 REPORT DATE October 1989	3 REPORT TYPE AND DATES COVERED Final
4 TITLE AND SUBTITLE OPTIMAL SELECTION THEORY FOR SUPERCONCURRENCY		5 FUNDING NUMBERS WU: OMN
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 TD 1669
9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Chief of Naval Research Code ONT-20P Office of Naval Technology Arlington, VA 22217-5000		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 a mathematical programming approach to finding an optimal, heterogeneous suite of processors to solve supercomputing problems. This technique, called superconcurrency, 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.

14 SUBJECT TERMS superconcurrency code profiling optimal selection	supercomputing benchmarking Amdahl's Law		15 NUMBER OF PAGES 9 16 PRICE CODE
17 SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18 SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19 SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20 LIMITATION OF ABSTRACT Unlimited

Optimal Selection Theory for Superconcurrency

Richard F. Freund, Naval Ocean Systems Center

Abstract. This paper describes a mathematical programming approach to finding an optimal, hetereogeneous suite of processors to solve supercomputing problems. This technique, called superconcurrency, 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.

Keywords. Superconcurrency, Supercomputing, Code Profiling, Benchmarking, Optimal Selection, Amdahl's Law

1. Objective

Ercegovac [1988] has recently looked at the feasibility of a suite of heterogeneous processors to solve supercomputing problems. Resnikoff [1987] and Kamen [1989] have examined the cost-effectiveness of supercomputers (one generally finds the smaller mini-supers to be more cost-effective than the largest-sized processors). Bokhari [1988] 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 [1967] 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.

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 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 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 formulation also rests on two mathematical assumptions outlined below. 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.

2. Mathematical Formulation

Let us 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 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 $s_j \cap s_k = \emptyset$ if $j \neq k$. The choice of the s_j defines the code profiling and analytical benchmarking problem. We will shortly look at motivation and explanation of this point. We denote $\mathbf{C}$ 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 $\mathbf{I}$ 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 indexed configuration). Then the mathematical programming problem can now be stated as:

(1) MINIMIZE

$$T(x_i, s_j) = \sum_{i \varepsilon \mathbf{I}, j} t_i(s_j) / v_i$$

such that $\sum_i v_i \; c_i \leq \mathbf{C}$

Because of assumptions we will be able to make later about linearity, we will want to group together all code subsegments of the same type, e.g., vectorizable code subsegements, or ones susceptible to parallelism. Thus we will divide $\mathbf{S}$ into equivalence classes, σ_j, by code type. For convenience we will index these classes by the first subsegment of that type, e.g., $\sigma_1 = \{ s_j | s_1 \cong s_j$, where $\cong$ denotes having same code type$\}$. We will later use the notation $t_i(\sigma_k)$ to denote $t_i(\sigma_k) = \sum t_i(s_j)$ for all $s_j \, \varepsilon \, \sigma_k$.

3. Vector-only Example

Let us consider first an intuitive example using primarily vectorizable code. Typically some portion of the code will be decomposable and some non-decomposable. By decomposable vector code we mean vectorizable code for which the original problem can be broken up into some small number of independent vectorizable subcodes. Let us consider a specific code example. Suppose it to be 50% decomposable vectorizable and 50% non-decomposable vectorizable. Suppose further that the overall cost constraint is \$2M and that we have a choice of three machines. Machine x_1 costs \$2M and speeds up vectorizable code (over some baseline scalar system) by a factor of 5. Machine x_2 costs \$1M and speeds up vectorizable code by a factor of of 4. Finally machine x_3 costs \$1/3M and speeds up vectorizable code by 3. We can enumerate the possible solutions that satisfy the cost constraint, namely:

SOLUTION	SUITE	$\mathbf{P}$
1	$1 \; x_1$	5.00
2	$2 \; x_2$	5.33
3	$1 \; x_2 + 3 \; x_3$	6.09
4	$6 \; x_3$	5.14

Solution 1 clearly gives an overall speed-up factor (on the total vectorizable code) of $\mathbf{P} = 5$. Solution 3 gives a speed-up factor of 4 on the non-decomposable vector portion (on which we use machine x_2) and a speedup of a little better than 12 (12.8) on the decomposable portion. The time relative to the original scalar baseline is 1/2 * 1/4 for the half of the vector code that is non-decomposable. Assuming the decomposable code is distributed evenly over all four machines, the time for the other half of the code is 1/2 * 1/4 * (.75 * 1/3 + .25 * 1/4). Summing these two yields a total time relative to the original scalar baseline time of 1/8 + 5/128 or 21/128. Thus it has an overall speed-up factor of $\mathbf{P}$ = 6.09 (128/21). Similarly we can compute that for solution 2, $\mathbf{P}$ = 5.33 (16/3) because the total time relative to the original scalar baseline is 1/2 * 1/4 + 1/2 * (1/2 * 1/4) = 3/16. For solution 4 the relative total time is 1/2 * 1/3 + 1/2 * (1/6 * 1/3) so $\mathbf{P}$ = 5.14 (36/7). Thus solution 3 is optimal.

4. Underlying Assumptions

One of the fundamental performance limiting factors of vector or parallel supercomputers has been the performance characteristics of such architectures on codes and algorithms for which they are not well designed. This is because such machines have generally been used on entire codes and large codes usually have significant portions which are not vectorizable or portions which are not parallel. The underlying motivation for this paper is the desire to understand how to map the various atomic code portions to the types of architectures for which they are best suited. Three of the resulting needs that follow from this are a distributed intelligent network system (DINS) to control and schedule such code/architecture matching, and new kinds of analytical benchmarking and code profiling (briefly discussed below).

There are two fundamental mathematical assumptions used in this paper. One is the linearity of each machine's characteristic time function, t_i, i.e., $t_i(s_j)$ is assumed to be linear in code segment length $|s_j|$ (or implied length due to, say, DO loops). Actually this is normally an invalid assumption when considering the relationship of machine behavior across code types! However we will be making this assumption only when considering the effect of any machine type against optimally matched code. This is equivalent to saying that each type of architecture acts linearly (asymptotically) on that type of code for which it is best suited, e.g., a vector machine takes twice as long to process a vector of length 1000 as of length 500. It is precisely because to this assumption of linearity that we can legitimately refer to $t_i(\sigma_j)$ as well as $t_i(S_j)$. Similarly we will assume the decrease in time due to multiple copies of the same machine on optimally matched and decomposable code to be linear.

5. Mathematical Reformulation

Since we are supposing all the time functions, t_i, to be linear, we can write each such function as:

$$t_i(\sigma_j) = \alpha_i t_0(\sigma_j) + \beta_i$$

where the β_i are overhead factors the α_i are speed up factors (thus α_i are such that typically $0 \leq \alpha_i <<$ 1, and t_0 represents some baseline time function such as that derived from, say, a VAX. Since we are primarily interested in asymptotic results, we will ignore the β_i and simplify the expression to be:

$$t_i(\sigma_j) = \alpha_i t_0(\sigma_j)$$

Now for any given code, S, let it (or at least its "hot spots") be divided into non-overlapping subsegment sets, σ_j. Let π_j be the percentage of time spent by the baseline machine on code subsegment set σ_j, i.e., $t_0(\sigma_j) = \pi_j$. Thus we have the baseline percentages given by:

$$T_0(S) = t_0\left(\sum_j \sigma_j\right) = \sum_j t_0(\sigma_j)$$

$$= \sum_j \pi_j = 1$$

Finally we need to deal with the speed-ups given in the decomposable cases (vector decomposable or all parallel types). As mentioned above we assume that for V processors this will give a linear speed-up of V, i.e., for V processors with characteristic time function, t_i, we will have total time given by t_i / v_i. It will be understood that v_i is 1 for non-decomposable cases. This gives us a reformulation of the original mathematical program as:

(2) MINIMIZE

$$T(x_i, \sigma_j) = \sum_{i \in I, j} t_i(\sigma_j)/v_i$$

$$= \sum_{i \in I, j} \alpha_i \pi_j / v_i$$

such that $\sum_i v_i c_i \leq C$

(where v_i may be 0 if machine x_i not in optimal configuration)

6. Multi-type Fxample

Let us consider a more complicated example than earlier. In this we shall have different types of vector, scalar, and parallel subcode. We will suppose we have code that is 50% vectorizable (35% non-decomposable and 15% decomposable), 20% fine-grain parallel, 20% coarse-grain parallel, and 10% scalar. For each of the possible machines below, we shall give speed-up factors on code for which they are designed, as well as scalar speed-up (over some baseline), in case the machine is used for scalar code. We shall also 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. Suppose our overall cost constraint is \$4M. Let our vector machines be: x_1 with cost \$4M, vector speed up of 10, and scalar speed-up of 2, x_2 with cost \$1M, vector speed-up of 5, *and scalar speed-up of 2, and x_3 with cost \$1/3M, vector* speed-up of 3, and scalar speed-up of 1. Let our fine-grain parallel machines be: x_4 with cost \$1M, parallel speed-up of 15, x_5 with cost \$1/3M, parallel speed-up of 6, and scalar speed-up of 1. Let our coarse-grain parallel machines be: x_6 with cost \$1M, parallel speed-up of 4, scalar speed-up of 1, and x_7 with cost \$1/3M, parallel speed-up of 2, and scalar speed-up of 1. Finally let x_8 be a scalar machine with cost \$1/4M and speed-up of 2.

One solution would be to spend \$4M on the highest performing vector machine, x_1, and in fact this is the traditonal supercomputing solution. This solution gives a speed-up factor of 10 on the vector portion and a speed-up of 2 on the rest. Thus its total time relative to the original scalar baseline is 1/20 + 1/4 = 6/20. Thus it has an overall speed-up factor of P = 3.33. This solution is not nearly optimal! Consider a solution of one x_2, three x_3, one x_4, and one x_6. This solution has overall speed-up of 5 on the non-decomposable vector, better than 12 on the decomposable, 15 on the fine-grain, 4 on the coarse-grain, and 2 on the scalar. Thus its total time relative to the original scalar baseline is .35/5 + .15 * 1/4 * (.25 * 1/5 + .75 * 1/3) + .2/15 + .2/4 + .1/2. Thus it has an overall speed-up factor of P = 5.139. Furthermore this does not even consider the secondary advantage of this multi-machine solution that the machines not being used on this code at any one time are available for other work. This example is, we believe, representative of a wide class of supercomputing problems in which the best total speed-up comes from a multi-machine solution in which no one machine is a traditonal supercomputer.

7. Mixed Strategies

There is an extension of the mathematical program, (2), above when we want a mixed strategy of optimizing several project applications with varying priorities. Suppose there are Q projects, p_k, (k=1, . . . Q) with relative weightings λ_k and $\sum_k \lambda_k = 1$.

Where several application projects need to be accomodated by the optimum choice of a mix of processors, we may weight the times spent on the code profiles types in each project by the importance of that project and use the sums of these derivative times (again suitably calibrated to add up to 1) to determine the relative need for processors to handle different code profile types. For example, suppose there are just two projects, a high-priority one with weighting $\lambda_1 = 0.7$, and a low-priority one, with $\lambda_2 = 0.3$. If a particular profile type appears in just the high-priority project, the percentage of time that it takes up in the high-priority project can be weighted by 0.7, whereas if it appears also in the low-priority project, the percentage of time that profile type takes in the low-priority project weighted by 0.3 would be added to the original weighted value to give the overall value to contribute to minimizing time under the cost constraints.

Suppose there are M profile types, m_k, (k=1,...,M) with relative time requirements that are calibrated to sum to 1 for each project. Then we have an M x Q matrix to express the distribution of time needed to handle the code profile types for the projects and a Q x 1 matrix of importance values associated with the various projects which can be multiplied to produce an M x 1 matrix of weighted times for the different code profiles. These weighted times can then be used, once scaled to add to 1, as the π_j for the code profiles in (2) above.

If the priorities of the different projects are used in this way, the original assignment of the priorities to projects will need to take into account the quantity of code for the project, since the process of scaling of the times needed for code of different types is done by project, and distinctions of code volume among the projects are thereby erased. If there are Q projects each with some proportion of the total importance value of 1, we have for the jth code profile type:

$$\sigma_j = \sum_k \lambda_k * \pi_{j,k}$$

The new σ_j's can be scaled so that they add up to 1 to produce new π_j's to replace the old π_j's in equation (2). This revision will permit the evaluation of the optimum mix of processors to be sensitive to priorities favoring some projects. Note that "project" can be interpreted as "project application" if it is felt that within a project some applications have different priorities than others. With this interpretation, Q will just be a larger real number.

8. Code Types and Benchmarking

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 sort 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 a benchmark entire 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 [1989] on LINPACK and Murphy [1989] on Livermore Loops 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 small Connection Machine, DAP, Ardent, Encore, Butterfly, 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.

Bandwidth 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. This can always be done optimally with a superconcurrent approach but on an ad hoc basis with reduced theoretical value.

Distributed Intelligent Network System (DINS) - One of the most active current research areas of the SRT is DINS. DINS will be a reasoning system that uses information from Code Profiling, Analytical Benchmarking, and network bandwidth to optimally 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 that 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.

9. 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.

Acknowledgments

This research was supported by the Office of Naval Technology and the Naval Ocean Systems Center. In addition the author would like to thank Roxana Kamen for her help, support, and encouragement.

References

[1] M. ERCEGOVAC, *Heterogeneity in Supercomputer Architectures*, Parallel Computing, vol 7 (1988), pp. 367-372.
[2] H. L. RESNIKOFF, *Cost-Effectiveness of Concurrent Supercomputers*, Journal of Supercomputing, vol 1 (1987), pp. 231-262.
[3] S. H. BOKHARI, *Partitioning Problems in Parallel, Pipelined, and Distributed Computing*, IEEE Transactions on Computers, vol 37 (1988), pp. 48-57.
[4] G. M. AMDAHL, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capability*, Proc. AFIPS Comput. Conf., Vol 30, 1967
[5] R. B. KAMEN, *Comparisons of Supercomputer Costs and Peak Performance*, NOSC Technical Note 1579, May 1989
[6] J. J. DONGARRA, *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Envrionment, Argonne National Laboratory*, Technical Memorandun No. 23, February 12, 1989
[7] C. G. MURPHY, *Evaluation of Advanced Computer Architectures for Supercomputing*, Science Applications International Corporation, 1989