CMST 5(1) 7-19 (1999) DOI:10.12921/cmst.1999.05.01.07-19

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY 5, 7-19 (1999)

HIGH-PERFORMANCE COMPUTING ON HETEROGENEOUS SYSTEMS

WOJCIECH CENCEK

Quantum Chemistry Group, Department of Chemistry,
A. Mickiewicz University, Grunwaldzka 6, 60-780 Poznań, Poland

Abstract: A model two-processor heterogeneous computer consisting of one scalar and one vector processor is analyzed in terms of its performance. It is demonstrated that on mixed-type (scalar-vector) applications it is much more effective than a homogeneous environment. Various models of the job distribution are introduced. A working implementation in the field of quantum chemistry is presented.

1. INTRODUCTION

Parallel and distributed computing is currently one of the hottest topics in both computer sciences and industry. This clearly reflects the fact that the growth of our needs and expectations is even faster than the amazing progress in microprocessor technology witnessed by us during these years. The main obstacle in an effective design and use of multiprocessor machines is the complicated logic of such computing environment, which has to cope with interprocessor communication, synchronization, memory access conflicts and other problems. The ultimate goal, the Holy Graal of parallel computing, is the "Metacomputer" - a gigantic machine (perhaps even the whole Internet) which fully automatically takes the users' requests and allocates the jobs optimally using its distributed heterogeneous resources. Such machine would be able to parallelize the codes without any intervention of the users to whom it would seem to be a single, infinitely powerful processor. Unfortunately, we are still somewhere around the beginning of the long road to this comfortable situation. The parallelization of all but trivial tasks requires a considerable effort of the programmer and the efficiency (in terms of computing time reduction relative to the number of busy processors) is often quite low which signifies a waste of resources and makes the whole business questionable.

One way to use the resources more efficiently is the distribution of the computation among processors of different types, i. e. the *heterogeneous* rather than *homogeneous* environment. Intuitively it is obvious that two processors with comparable *average* power but different architectures perform differently depending on specific features of performed operations. Hence, some calculations run faster on the first processor while other are better suited for the second one. Assigning different parts of the job to different processors should then help to reduce the computation time. The aim of this paper is to yield a strict quantitative analysis of such situation. It will be assumed that each processor has its own private memory. This *distributed memory* model is more general than the *shared memory* because it leaves complete freedom in combining together individual processing units which can in fact be separate machines connected by a network. The only way of exchanging information between nodes is the explicit message

passing. For such structure the name "multicomputer" used for example by Foster [1] seems to be more appropriate than "multiprocessor". It will be also assumed throughout the paper that the time needed to send and receive messages is small compared with the calculations i. e. that the communication cost can be neglected. All our performance predictions should be therefore understood as upper bounds to what can be gained in real systems.

2. SCALAR AND VECTOR PROCESSING

When looking for criteria according to which a job should be distributed among different processors, the most obvious factor is how efficient the particular nodes are on scalar and on vector problems. Vector computations involve indexed variables and run efficiently on machines featuring specialized vector registers (like Cray X-MP, Y-MP, or C-90 series) or fast caching and pipeline mechanisms (like Intel i860). In scalar computations other factors - like faster CPU clocks - are more important. Although with the advent of modern superscalar RISC chips the traditional distinction between scalar and vector machines has been weakened, the speed of particular machines still depends strongly on the type of performed operations. Table I presents execution times of a scalar and a vector benchmark on the computers listed below. The bench-marks are taken from the real-life quantum chemistry code described in Sect. 5.

SGI RI0000 Silicon Graphics Origin 200, processor MIPS R10000 180 MHz
Operating system: IRIX64 6.4
Compiler: Mongoose Fortran 7.11
Compiling command: f77 -64 -mips4 -03
Libraries: BLAS (libblas.a)

SGI R8000 Silicon Graphics Indigo 2, processor MIPS R8000 75 MHz
Operating system: IRIX64 6.2
Compiler: Mongoose Fortran 7.10
Compiling command: f77 -64 -mips4 -03
Libraries: BLAS (libblas.a)

NS-860 Microway Number Smasher, processor Intel 860-XR40 MHz
Operating system: MSDOS 5.0
Compiler: Microway NDP Fortran-860 4.Id
Compiling command: mf860n -on
Libraries: Kuck & Associates Library (libkden.a, libkmath.a)

Pentium IBM PC compatible, processor Intel Pentium 100 MHz
Operating system: MSDOS 5.0
Compiler: Microway NDP Fortran-486 3.20
Compiling command: mf486 -n2 -n3 -on -OL
Libraries: none

Cray **EL** Cray Y-MP EL 33 MHz
Operating system: Unicos 8.0.4.2

Compiler: Cray Fortran CFT77 6.0.2.0
Compiling command: cf77 -dp -Ovector3 -Oscalar3
Libraries: Cray Research Scientific Library (libsci.a)

Cray J916 Cray J916 100 MHz
Operating system: Unicos 9.0.2.4
Compiler: Cray Fortran CFT77 6.0.4.24
Compiling command: cf77 -dp -Ovector3 -Oscalar3
Libraries: Cray Research Scientific Library (libsci.a)

Cray T3E Cray T3E-900, processors DEC Alpha 450 MHz
Operating system: Unicos/mk 1.5.2
Compiler: Cray Fortran CF90 3.0.1.0
Compiling command: f90 -dp -Oaggress,pipeline3,scalar3,vector3
Libraries: Cray Research Scientific Library (libsci.a)

HP 715 Hewlett-Packard 715, processor PA-RISC 50 MHz
Operating system: HP-UX 10.20
Compiler: HP Fortran/S700 10.20.02
Compiling command: f77 +03
Libraries: BLAS, Lapack (libblas.a, liblapack_hppa.a)

IBM SP2 IBM 9076 Scalable POWERparallel System, processors POWER2 66 MHz
Operating system: AIX 4.1.4
Compiler: AIX XL Fortran 03.02
Compiling command: f77 -03
Libraries: BLAS (libblas.a)

Table I. Times (in seconds) from the scalar (t_s) and vector (t_v) benchmark

Machine	t_s	Machine	t_v
Cray EL	1.20	Pentium	0.538
NS-860	0.916	HP 715	0.434
Cray J916	0.473	NS-860	0.172
Pentium	0.316	SGI R8000	0.0967
HP 715	0.225	IBM SP2	0.0866
SGI R8000	0.124	SGI R10000	0.0703
IBM SP2	0.116	Cray EL	0.0604
Cray T3E	0.0465	Cray T3E	0.0499
SGI R10000	0.0372	Cray J916	0.0395

What counts is the fact that the orders of computers (according to the growing speed) in both cases differ significantly. For example, Cray EL happens to be the slowest of all the computers (!)

in the scalar test, but among the fastest in the vector test. To assess quantitatively the relative performance of the machines we proceed as follows. In each of the possible pairs we assign the label "*S*" (scalar) to one of the machines and the label "*V*"(vector) to the other, which can be done in two ways, so that we have in total $2\cdot\binom{N}{2}$ ordered pairs for the *N* computers. For each ordered pair we calculate - using the numbers listed in Table I - two following quantities:

$$\delta_S = \frac{t_S(V)}{t_S(S)} \qquad \delta_V = \frac{t_V(S)}{t_V(V)} \tag{1}$$

where the label in parenthesis denotes the machine a given test was run on. δ_s. is the ratio of the scalar speed of *S* to the scalar speed of *V*. Similarly, δ_v denotes the ratio of the vector speed of *V* to the vector speed of *S*. All the calculated values of δ_s and δ_v are listed in Table II. Note that the product of each two numbers lying symmetrically with respect to the main diagonal is equal to 1 because this symmetry corresponds to the exchange of the computer labels in Eq. 1. The most interesting situation occurs if both δ_s and δ_v for a given ordered *(V, S)*-pair are greater than 1 (e. g., all the pairs where V is Cray J916 or Cray EL and *S* is a workstation). In such cases, the machine chosen as V is faster than *S* on the vector test and *simultaneously S* is faster than *V* on the scalar test. If we now imagine that the scalar and vector computations are not separate tasks but rather parts of one job, it is obvious that the job should be distributed in such way that a larger fraction of the vector part should be assigned to *V* and *vice versa*. Various models of such distributions are presented in Section 3, while Section 4 deals with performance predictions for these models.

Table II. The values of δ_v (upper) and δ_s (lower) for different pairs of computers

V: *S*:	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(1) Cray T3E	1.0 1.0	0.79 0.098	1.2 0.039	1.4 1.3	1.9 0.38	3.4 0.051	1.7 0.40	8.7 0.21	11 0.15
(2) Cray J916	1.3 10	1.0 1.0	1.5 0.39	1.8 13	2.4 3.8	4.4 0.52	2.2 4.1	11 2.1	14 1.5
(3) Cray EL	0.83 26	0.65 2.5	1.0 1.0	1.2 32	1.6 9.7	2.8 1.3	1.4 10	7.2 5.4	8.9 3.8
(4) SGI R10000	0.71 0.80	0.56 0.079	0.86 0.031	1.0 1.0	1.4 0.30	2.4 0.041	1.2 0.32	6.2 0.17	7.7 0.12
(5) SGI R8000	0.52 2.7	0.41 0.26	0.63 0.10	0.73 3.3	1.0 1.0	1.8 0.14	0.90 1.1	4.5 0.55	5.6 0.39
(6) NS-860	0.29 20	0.23 1.9	0.35 0.76	0.41 25	0.56 7.4	1.0 1.0	0.50 7.9	2.5 4.1	3.1 2.9
(7) IBM SP2	0.58 2.5	0.46 0.25	0.70 0.097	0.81 3.1	1.1 0.94	2.0 0.13	1.0 1.0	5.0 0.52	6.2 0.37
(8) HP 715	0.12 4.8	0.091 0.48	0.14 0.19	0.16 6.0	0.22 1.8	0.40 0.25	0.20 1.9	1.0 1.0	1.2 0.71
(9) Pentium	0.093 6.8	0.073 0.67	0.11 0.26	0.13 8.5	0.18 2.5	0.32 0.34	0.16 2.7	0.81 1.4	1.0 1.0

Obviously, the values of δ_s and δ_v in Table II have no universal meaning and would be different if other scalar and vector tests had been applied. While the speed of scalar calculations only slightly depends on the task, the vector performance is more sensitive to the vector length and other factors. Nevertheless, both the parameters are a useful measure of the relative performance of different computers, especially if one is able to recalculate them for a narrow class of applications one is interested in.

3. HETEROGENEOUS DISTRIBUTION MODELS

The discussion in this and the next section is restricted to heterogeneous systems consisting of two arbitrary machines S and V. Firstly, since we consider only two distinct types of parts with significantly different characters, all the main ideas and effects resulting from the heterogeneity can be described in terms of such a model. Secondly, it can always be generalized by treating S and V as subsystems consisting of numbers of processors. We will call S scalar and V vector processor, but their actual performance does not affect our discussion and one should treat S and V just as labels.

Let us imagine a computer job which is a sequence of scalar (s_1, s_2,...) and vector (v_1, v_2,...) parts. These parts can be represented graphically as rectangles whose areas are proportional to the number of operations in the given parts. The rectangles are placed one after another along the horizontal axis denoting the elapsing time. The height of each rectangle therefore corresponds directly to the speed of the computation in the particular part. To illustrate these rules, Fig. 1 presents the job executed without any distribution, i. e. either on S or on V. To facilitate further discussion, this particular example is constructed in such a way that the total execution times on both machines, $t(S)$ and $t(V)$, are equal and amount to 10 seconds. This restriction will be removed in the next section, where a general definition of the speedup on a heterogeneous system will be introduced. Note that the label in the rectangle indicates that the given part of the job is executed on the pertinent machine. Let t_s (t_v) denote the overall time spent by a given machine in all scalar (vector) parts when the job is not distributed. Then

$$\begin{aligned} t(S) &= t_s(S) + t_v(S) , \\ t(V) &= t_s(V) + t_v(V) . \end{aligned} \tag{2}$$

The simplest possibility, which can be used even when none of the parts s_i and v_i is parallelizable, consists in assigning each part entirely to the processor that executes it faster, for example the scalar parts to S and the vector parts to V, as seen on the left plot of Fig. 2. We will call this **Model 1.** Its two main features are *full specialization* (each processor does only what it can do best) and *zero concurrence* (at no moment both processors run in parallel). The shortest time needed to finish the whole job on the system is clearly

$$t = t_s(S) + t_v(V) . \tag{3}$$

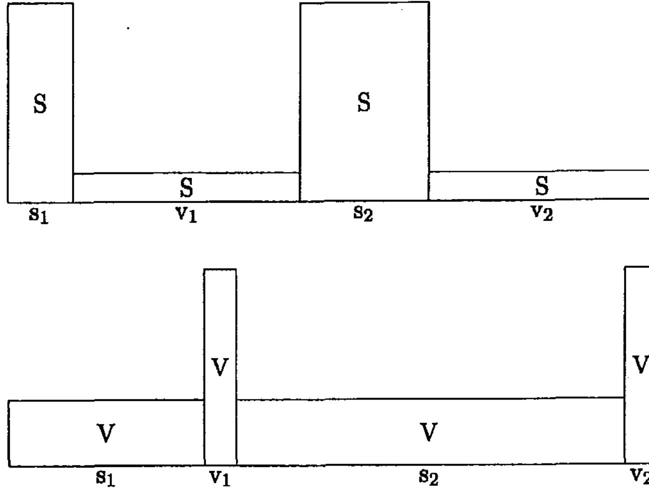


Fig. 1. A job running sequentially on a scalar (upper) machine S or a vector (lower) machine V

If, for example, $t_s(S) = 3$ seconds, $t_v(S) = 7$ seconds, $t_s(V) = 9$ seconds and $t_v(V) = 1$ second, (note that the times correspond precisely to our figures) then each of the two machines alone requires 10 seconds and the best time achievable by the whole system is only 4 seconds. This simple example illustrates two important facts: that the *superlinear speedup* (larger than the number of processors used - here 2.5 times faster on 2 processors) is not unusual on heterogeneous systems and that concurrence is not the only way to speed up the program execution. The reason is that the distribution of the job avoids the bottlenecks present in a single machine (the vector parts on S and the scalar parts on V).

One step further would be to parallelize the scalar parts leaving the vector ones to be executed sequentially on V (as seen on the right plot of Fig. 2) or *vice versa.* In this scheme, which we will call **Model** 2, processor V helps to execute the scalar parts which can be therefore finished faster than in Model 1. If the work is well load-balanced, i. e. each processor is assigned the fraction of each s_i which is proportional to its speed, then both finish the s_i's at the same time. The speed (reciprocal of time) of the heterogeneous system at the scalar parts is then the sum of the component speeds:

$$t_s^{-1} = t_s(S)^{-1} + t_s(V)^{-1} \tag{4}$$

and the total execution time is

$$t = [t_s(S)^{-1} + t_s(V)^{-1}]^{-1} + t_v(V) \,. \tag{5}$$

Using the same values of $t_s(S)$, $t_s(V)$ $t_v(S)$ and $t_v(V)$ as in the previous example we get in this case $t = 3.25$ seconds. The improvement is achieved thanks to a significant degree of concurrence, though the specialization is not complete. The analogous reasoning can be made in the case where only the vector parts instead of the scalar parts are parallelized.

Fig. 2. Model 1 (left) and Model 2 (right) of the distribution

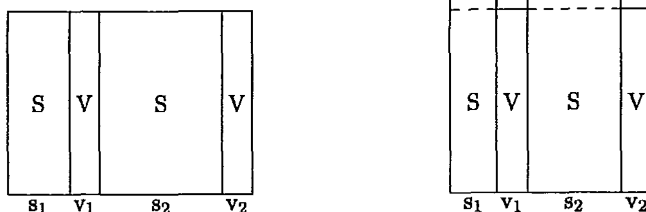
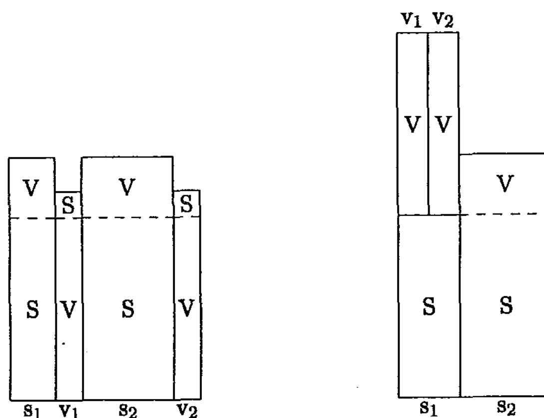


Fig. 3. Model 3 (left) and Model 4 (right) of the distribution



Note that the diagrams in Fig. 2 and in subsequent figures in this section can easily be constructed with the correct width (i. e., the correct total execution time) by complying with two simple rules. Firstly, the area of each s_i and v_i rectangle (number of operations) remains the same as in the sequential executions (Fig. 1). Secondly, the height of each sub-rectangle (the processor's speed), e. g. "*V*" in s_1"-subrectangle, is the same as the height of the corresponding rectangle in Fig. 1.

Still better performance is possible if all the parts of our job are parallelized, as presented on the left plot of Fig. 3. Ensuring good load balance also in the vector parts one gets

$$t_v^{-1} = t_v(S)^{-1} + t_v(V)^{-1} \tag{6}$$

and

$$t = [t_s(S)^{-1} + t_s(V)^{-1}]^{-1} + [t_v(S)^{-1} + t_v(V)^{-1}]^{-1}$$

which amounts in our example to 3.125 seconds. This scheme **(Model 3)** exhibits full concurrence (neither of the two processors is ever idle) but only partial specialization.

The degree of specialization can be augmented without destroying concurrence only at a coarser granularity of parallelism, i. e. if the scalar parts could be done in parallel with the vector ones **(Model 4),** as illustrated on the right plot of Fig. 3. If $t_s(S)$ happens to be equal to $t_v(V)$, the scalar and vector parts are finished simultaneously, each done exclusively by "its" processor (full concurrence + full specialization). The execution time then is $t = t_s(S) = t_v(V)$. If

one of the processors finishes its work first (like V in Fig. 3), it takes over the appropriate fraction of the work left to the second processor. In this case t can be expressed as

$$t = |t_s(S) - t_v(V)| \begin{cases} \times \dfrac{t_s(S)^{-1}}{t_s(V)^{-1} + t_s(S)^{-1}} + t_v(V) \text{ if } t_s(S) > t_v(V) \\ \times \dfrac{t_v(V)^{-1}}{t_v(S)^{-1} + t_v(V)^{-1}} + t_s(S) \text{ if } t_s(S) < t_v(V) \end{cases} \quad (8)$$

and amounts to only 2.5 seconds in our example. However, Model 4 is seldom applicable, because it requires the order of computations to be changed, which is possible only if vector and scalar parts are mutually independent. In real jobs calculations of the scalar parts will most likely depend on the results of the vector parts and *vice versa,* which in our example means that, for example, v_1 cannot start before s_1 has been finished.

4. SPEEDUP ON A HETEROGENEOUS SYSTEM

In order to compare quantitatively the behaviour of all the models with one another and with homogeneous systems it is necessary to introduce an appropriate performance measure, such as *speedup.* For the case of a homogeneous multiprocessor system, an obvious and well-known speedup definition is

$$S = \frac{t_1}{t}, \quad (9)$$

where t is the execution time on the system and t_1 on the single processor. A difficulty arises immediately in the heterogeneous case, since t_1 is, in general, not uniquely defined and depends on the processor on which it is measured. In fact, the very notion of the "speedup" is far from being strict in this situation. From the economical point of view, it is reasonable to understand the speedup - rather restrictively - as the performance improvement relative to the best single-processor situation, i. e. to choose

$$S = \frac{\min\{t_i\}}{t} \quad (10)$$

as suggested, for example, in Refs. [2] and [3], In our two-processor models we have

$$S = \frac{\min\{t(S), t(V)\}}{t}. \quad (11)$$

By putting into (11) the expressions for $t(S)$ and $t(V)$ from (2) and for t from (3), (5), (7), or (8), one can obtain the speedup S in each model as functions of 4 variables: $t_s(S)$, $t_v(S)$, $t_s(V)$ and $t_v(V)$. These functions are homogeneous of the order zero, i. e. invariant with respect to the simultaneous multiplication of all the variables by the same constant (which is rather obvious, because the speedup does not depend on the units in which all the times are expressed). This

means that 3 variables are also sufficient, the simplest choice being to set e. g. $t_s(S)$ to 1 and to express the others in units of $t_s(S)$. However, each of these variables depends on the properties of both the job and the processor and such functions would not be easy to interpret. A more useful set is to take δ_s and δ_v from Eqs. 1, which depend only on the employed computer pair, and the third parameter

$$x_s = \frac{t_s(S)}{t_v(S)} \ . \tag{12}$$

which characterizes the job and takes values from $x_s = 0$ (purely vector jobs) up to $x_v = \infty$ (purely scalar jobs). Its concrete value depends also on the machine that plays the role of S, but for each choice of the (S, V)-pair it orders uniquely all the possible jobs according to the relative contribution of the vector and scalar operations. The speedup can now be expressed in terms of δ_s; δ_v, and x_s for each of the distribution models. In the following equations, $D = \delta_v\delta_s x_s - \delta_v x_s - \delta_v + 1$.

$$1: \quad S = \begin{cases} \delta_v(x_s + 1)(\delta_v x_s + 1)^{-1} & \text{if } D \geq 0 \\ (\delta_v\delta_s x_s + 1)(\delta_v x_s + 1)^{-1} & \text{if } D \leq 0 \end{cases} \tag{13}$$

$$2: \quad S = \begin{cases} \delta_v(\delta_s + 1)(x_s + 1)(\delta_v\delta_s x_s + \delta_s + 1)^{-1} & \text{if } D \geq 0 \\ (\delta_s + 1)(\delta_v\delta_s x_s + 1)(\delta_v\delta_s x_s + \delta_s + 1)^{-1} & \text{if } D \leq 0 \end{cases} \tag{14}$$

$$3: \quad S = \begin{cases} (\delta_v + 1)(\delta_s + 1)(x_s + 1)[(\delta_v + 1)\delta_s x_s + \delta_s + 1]^{-1} & \text{if } D \geq 0 \\ (\delta_v + 1)(\delta_s + 1)(\delta_v\delta_s x_s + 1)[\delta_v(\delta_v + 1)\delta_s x_s + \delta_s + 1]^{-1} & \text{if } D \leq 0 \end{cases} \tag{15}$$

$$4: \quad S = \begin{cases} \delta_v(\delta_s + 1)(x_s + 1)[\delta_s\delta_v x_s + 1]^{-1} & \text{if } D \geq 0 \text{ and } \delta_v x_s \geq 1 \\ \delta_v + 1 & \text{if } D \geq 0 \text{ and } \delta_v x_s \leq 1 \\ \delta_s + 1 & \text{if } D \leq 0 \text{ and } \delta_v x_s \geq 1 \\ (\delta_v + 1)(\delta_s\delta_v x_s + 1)[\delta_v(x_s + 1)]^{-1} & \text{if } D \leq 0 \text{ and } \delta_v x_s \leq 1 \end{cases} \ . \tag{16}$$

An interesting question is: What is the highest possible speedup S_{max} (i. e. that achievable if the proportion of scalar to vector operations in the job is optimal) for a given pair of computers? To obtain the answer one has to solve the equations $\partial S/\partial x_s = 0$ for x_s, which is straightforward but must be done with some care due to the interval definitions of S.

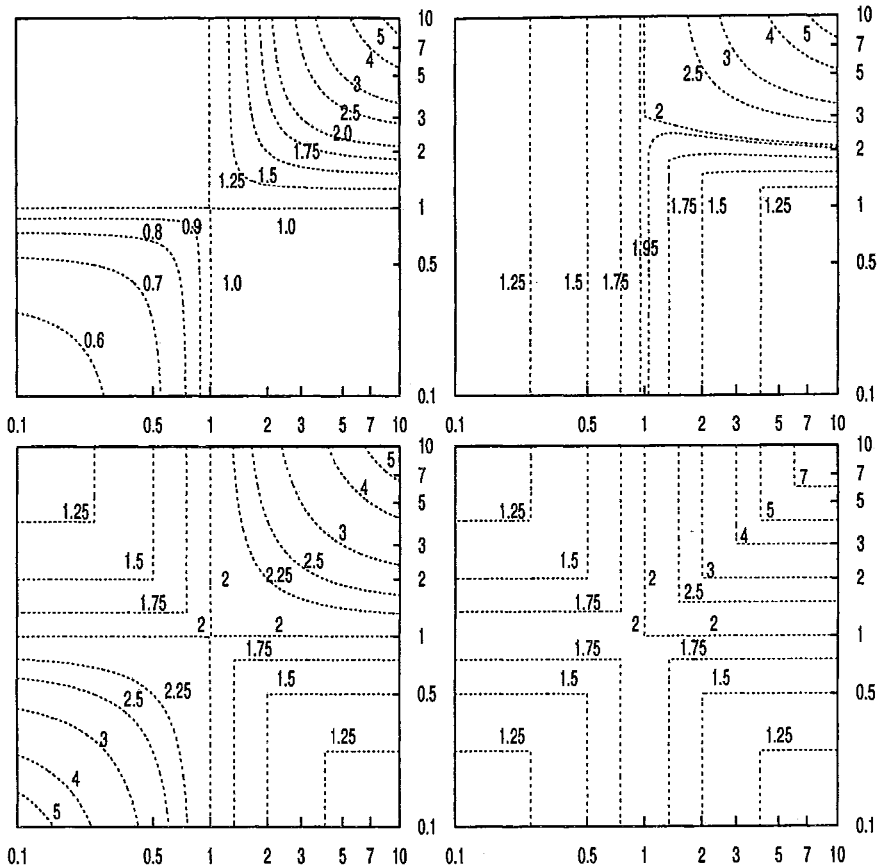


Fig. 4. Maximal speedup S_{max} achievable in Model 1 (upper left), Model 2 (upper right), Model 3 (lower left), and Model 4 (lower right) as a function of δ_s (horizontal axes) and δ_v (vertical axes)

S_{max} computed as a function of δ_s and δ_v is plotted in Fig. 4. In general, high speedup values are possible **if both δ_v and δ_s are large** (the upper right corners of the plots). This is easy to understand, because large δ_v and δ_s. mean that each processor is significantly faster than its partner at this part of the job this processor is mainly assigned to. In Model 3 the lower left comer is equally good, because this model is invariant with respect to the exchange *(V ↔ S),* or equivalently ($\delta_s \to \delta_s^{-1}$ $\delta_v \to \delta_v^{-1}$), which can be compensated by changing distribution ratios (relative areas of the *S*- and V-sub-rectangles in the left plot of Fig. 3), whereas in the other three models the label *Vis* always ascribed to the processor which alone computes the vector parts v_i.

The graduate performance improvement when going from Model 1 up to Model 4 is also nicely seen as the movement of contours with given values of S_{max} from the plots' upper right corners towards the centers. In Model 1 (without concurrence!) quite large values are necessary to obtain the superlinear (greater than 2) speedup. However, as seen on the first plot in Fig. *4,* any pair of computers with $\delta_v > 1$ and $\delta_s > 1$ can execute the job faster than on the single processor,

which is especially valuable if the parallelization of the individual vector and (or) scalar parts of the job is not possible or prohibitively complicated. In the last two models, any such pair is already sufficient to obtain (at some class of jobs - remember the role of x_s) the superlinear speedup.

The main message from our analysis is that heterogeneous computing on an appropriate combination of scalar and vector computers can be much more effective than homogeneous computing. Remember that Fig. 4 illustrates just the two-processor configuration, where the speedup on homogeneous machines cannot exceed 2!

5. EXAMPLE IMPLEMENTATION

The ideas outlined in the previous sections can be applied to any computational problem fulfilling the following two criteria. Firstly, it should consist of distinctive scalar and vector parts. Secondly, the amount of scalar and vector operations to be done should be of similar orders of magnitude if one expects to gain really something from heterogeneity. In fact, such mixed-type applications are very common in practice. We will focus now on one example taken from the work of our Quantum Chemistry Group at the Adam Mickiewicz University [4, 5].

The central problem of quantum chemistry consists in solving the molecular (or atomic) Schrödinger equation

$$\hat{H}\Psi = E\Psi \tag{17}$$

where $\hat{H}$ is a linear operator defined for each quantum system. The total energy of the molecule, E, and its so-called wave function, Ψ, are to be found. The knowledge of Ψ allows one to calculate in a straight-forward way all static properties of the molecule. However, analytical solutions of Eq. (17) are not known and one has to look for approximations. In the most general linear variational method, the unknown wave function Ψ is expanded as a combination of some basis functions Φ_i

$$\Psi = \sum_{i=1}^{K} c_i \phi_i \tag{18}$$

which leads to the linear matrix equation for the vector c of the best parameters c_i

$$\boldsymbol{Hc} = E\boldsymbol{Sc} \,. \tag{19}$$

In the above equation, $\boldsymbol{H}$ and $\boldsymbol{S}$ are matrices built up of integrals computed with the basis functions

$$h_{ij} = \int \phi_i \hat{H} \phi_j dV \tag{20}$$

$$s_{ij} = \int \phi_i \phi_j dV \tag{21}$$

and dV stands for the integration over the full domain. It can be shown that the approximate value of E resulting from (19) is always higher than the true solution of (17). This yields a criterion of

the good choice of the basis functions Φ_i in Eq. (18): The computed value of E should be as low as possible. If the Φ_i's depend on some parameters, it is possible to optimize them looking for the minimum of the energy. These variations are, however, nonlinear and the minimum is found in laborious iterations rather than from a closed equation. The computational scheme of the search can be written as follows:

1) Choose starting values of the nonlinear parameters in $\{\Phi_i\}$.
2) Solve the matrix equation (19).
3) Modify the nonlinear parameters.
4) Compute the new values of the matrix elements (20) and (21).
5) Goto 2.

In practice, steps 2-5 have to be repeated thousands or millions of times to make the nonlinear parameters converge. The modifications in step 3 (with negligible execution time) take place in only one basis function at a time, therefore only one row of each matrix needs to be updated in step 4. Step 4 involves long sequences of floating point additions, multiplications, divisions, and calls to the square root and exponential functions and is a typical scalar problem. Time needed to compute 512 pairs h_{ij} and s_{ij} for a typical three-electron molecule is what we used as the scalar benchmark t_s in Table I. Step 2 is purely vector and can be coded as calls to standard Lapack and BLAS routines. t_v in Table I corresponds to the solution of Eq. 19 of the order 512. One can see in Table I that both the times are of the same orders of magnitude which means that distribution on a heterogeneous system should be particularly efficient. So far, we have developed only Model 1 and Model 2 distributions of the code and implemented them on the system composed of Pentium 100 Mhz and Number Smasher machines (see descriptions in Sect. 2). Obviously, both components are rather outdated compared with what is currently available. However, this is an ideal pair for test purposes because of its low cost and the fact that both parameters $\delta_s = 2.9$ and $\delta_v = 3.1$ are relatively large (see Table II). By putting these values in Eqs. (13) and (14) and maximizing S with respect to δ_s one obtains $S_{max} = 1.98$ in Model 1 and $S_{max} = 2.30$ in Model 2, which can also be approximately found from Fig. 4. In real calculations, the highest speedup in Model 1, $S = 1.50$ was obtained in the optimization of a 150-term wave function of the H_2 molecule ($K = 150$ in Eq. 18), and in Model 2, $S = 2.13$, for a 600-term wave function of the He_2^+ molecule. These speedups are lower than predicted for two reasons. Firstly, the predicted S_{max} correspond actually to particular jobs where the fraction of scalar and vector operations is optimal (see the discussion in Sect. 4). Secondly, the predictions neglect the communication cost. However, it is nice to see the confirmation of the superlinear speedup in Model 2, a phenomenon which could not be possible without the appropriate use of the heterogeneous environment.

6. CONCLUSIONS

The calculations reported in the previous section are rather preliminary but clearly confirm that the heterogeneous environment allows one to use computational resources very efficiently. Already in Model 2, which is very simple and leaves half of the job not parallelized, the superlinear speedup (larger than the number of processors) could be obtained.

Looking at Table 2 one can see that the Pentium-860 pair is not the only (and in fact not the best) combination for building heterogeneous systems. Significantly larger values of both δ_v and δ_s are exhibited by some pairs containing vector Cray processors. A large value of δ_s in such cases means that by running a mixed scalar-vector application on the Cray processor one does not use its power efficiently because the scalar parts of the job are a bottleneck and could be executed several times faster by assigning them to some other (scalar) processor. It would be, perhaps, an interesting idea to build workstations containing at least one vector processor and at least one scalar chip, with a fast internal communication ensuring independence from the external network load.

Acknowledgments

The author thanks Professor J. Rychlewski for stimulating discussions. This work was supported by the KBN grants 8 T1 IF 00712 and SPUB/COST-D9.

References

[1] I. Foster, *Designing and Building Parallel Programs* (online) (1995). Available at http://www.mcs.anl.gov/dbpp/.

[2] C. R. Mechoso, J. D. Farrara, J. A. Spahr, *Achieving superlinear speedup on a heterogeneous, distributed system*, IEEE Parallel & Distributed Technology **2,** 57 (1994).

[3] X. Zhang and Y. Yan, *Modeling and characterizing parallel computing performance on heterogeneous networks of workstations*, Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing 25 (1995).

[4] W. Cencek and J. Rychlewski, *Many-electron explicitly correlated Gaussian functions*. I. *General theory and test results*, J. Chem. Phys. **102,** 2533 (1995).

[5] W. Cencek, J. Komasa, J. Rychlewski, *Benchmark calculations for two-electron systems using explicitly correlated Gaussian functions*, Chem. Phys. Lett. 246, 417 (1995).