A Practical Method for Load Balancing using Newton's Method on MPP Systems

Brent Swartz, Lockheed Martin Technical Services, Inc., Bay City, MI, USA

ABSTRACT: *A general method is presented for load balancing an application over a given number of Massively Parallel Processing (MPP) or Symmetric Multi-Processing (SMP) nodes* when *a function can be found which represents the amount of time (compute work) required for program completion as a function of the data domain. The method determines which data needs to be distributed to each node (i.e. determines the domain decomposition) such that each node finishes at precisely the same time. The procedure reduces the load balancing problem to one of finding this function for a given application.*

1 Introduction

Increasingly complex problems are being solved on parallel processors, such as Symmetric Multi-Processors (SMPs) and Massively Parallel Processors (MPPs). Decomposing the domain of a program, i.e. assigning the data associated with the program to the processors which will be running the program, is not always trivial. For instance, load balancing becomes a problem when the amount of compute work assigned to the processors is not equal, so that some processors are forced to be idle while they wait for other processors, the ones with too much work, to catch up with them at synchronization points within the program. If the amount of compute work associated with a given datum in the domain of the program is constant, then the load balancing problem becomes easier because the program becomes load balanced when equal amounts of the domain are distributed to the processors that are available. However, when programs have variable amounts of compute work for different parts of the domain, another method for distributing the work amongst the processors must be found to minimize the amount of idle time.

This paper describes a method for distributing a program's domain so that it will be perfectly load balanced when a function can be found which represents the amount of compute work (or time) required as a function of the data domain. The second section of this paper describes the theory on which this procedure is based. The third section describes practical details of implementation by application of the method in one dimension. Section four generalizes the method to heterogeneous nodes. The fifth section discusses the application of this method to the multi-dimensional case, and section six discusses the possibility of automating this method by incorporating it into a code parallelization tool. Section seven summarizes conclusions. The appendices discuss the conditions under which Newton's method is guaranteed to converge, since this method can utilize Newton's method (if the function is differentiable).

2 Theory

The basic method is described using the following definitions:

N_p is the number of processors (nodes),
x is the program's data domain,
x_i is the lower bound of the portion of x allocated to node i, which is also the upper bound of node i-1,
$t(x)$ is the function representing the cumulative time required for solution as a function of domain x,
TT is the total time, the total amount of time required for program completion, and
x_{max} is the maximum value of x for this run of the program
i is the node number, $0 <= i < N_p$

In this treatment:

- $t(x)$ is exact. The closer $t(x)$ approximates reality, the better the load balancing efficiency should be.

- Communication time is not taken into account.
- "Program" can be defined as any task executed between synchronization points.

With these definitions the procedure is now described:

In order to perfectly load balance an application across N_p nodes, each node must do exactly the same amount of work, i.e., finish it's task in the same amount of time TT/N_p. In order to utilize this method a cumulative time function $t(x)$ is needed to map the amount of time consumed by the program as a function of the domain x (generally the grid space over which the program is iterating). For simplicity the one dimensional case is considered here, and the multi-dimensional case is discussed in Section V.

Each node can be allocated an equal amount of the work (time). As depicted graphically in Figure 1, this is equivalent to subdividing the t-axis into equally sized adjacent regions of size TT/N_p. A root finding method can then determine the x_i coordinate corresponding to the region end points, as shown below. By definition, in the one-dimensional case, assuming $t(x)$ is exact, $TT = t(x_{max}) = t(x_{Np})$. Assigning each processor the same slice of work
(TT/N_p), $t(x_i) = i * TT/N_p$, or, for $0 <= i <= N_p$,

$$t(x_i) - i * TT/N_p = 0 \qquad \textbf{(1)}$$

So determining the domain x_i that processor i should work on has been reduced to finding the roots of the function $t(x_i) - i * TT/N_p$, and the domain of node i will then be $[x_i, x_{i+1}]$. Note that each node can determine it's domain independently of the other nodes, so that only two calls to a root finding routine are required on each node. To ensure there is no overlap of the domain on the nodes, a node's domain can be defined as $[x_i, x_{i+1}-\varepsilon]$, where ε is the smallest machine representable change in x. If equation (1) can be solved analytically for x, then this explicit formula for x can be used to distribute the domain x across N_p nodes. This could possibly be implemented with High Performance Fortran version 2 (HPF2) user defined distribution functions. If this function is reasonably well-behaved, then Newton's method can be utilized to find the part of the domain that a given PE should work on to achieve good load balance. If possible, the analytic form for $t'(x)$ can be used, otherwise $t'(x)$ can be determined with a discretezation such as
$t'(x) = dt/dx = (t(x+dx)-t(x))/dx$
(but care must be used in the choice of dx here).

Once $t(x)$ and $t'(x)$ have been found, it is straightforward to use the Newton-Raphson Method to rapidly determine x_i exactly, assuming the behavior of these functions is reasonable (i.e. $t(x)$ is continuous and differentiable). If the function's behavior is <u>not</u> reasonable, then another method, such as the secant or bisection method, can be utilized to find the roots. Newton's Method is preferred because of it's superior convergence properties, since this domain distribution algorithm constitutes overhead for the program. As shown before this will require a maximum of 2 $t(x)$ evaluations per step of Newton's Method ($t(x)$ and $t(x+dx)$). See Appendix A for a discussion of the convergence of Newton's method to the root.

If $t(x)$ is not known a priori, then good results may be obtained by curve fitting an appropriate well-behaved function to sample timings. The $t(x)$ function has at least one constant which must be determined empirically because every computer has different performance characteristics that will affect $t(x)$. Fortunately, this constant divides out (this is shown in the example at the end of section V). Writing $t(x)=A*t_{ref}(x)$, where A is the machine dependent constant, and knowing that $TT=t(x_{Np})$, Equation (1) can be divided by A to get a generic, portable equation which can be solved for x_i:

$$t_{ref}(x_i) - i * t_{ref}(x_{max})/N_p = 0$$

Thus only the <u>form</u> of $t(x)$ is required to use this method.

Figure 2 depicts the amount of compute/idle time for a sample program run, which motivates the definition of a measure of a program's load balance. The Load Balance Inefficiency (L_I) is defined to be:

$$L_I = (T_{max} - T_{avg})/T_{avg} * 100 \qquad \textbf{(2)}$$

and the Load Balance Efficiency (L_E) to be:

$$L_E = 100 - L_I \qquad \textbf{(3)}$$

where T_{max} is the maximum time to completion for any node in the partition and
T_{avg} is the average time to completion for all nodes in the partition.

The L_I indicates how much computer time was wasted due to load imbalance as a percentage of the total amount of time (T_{avg}) that a perfectly load balanced program would have taken. Each node is idle from the time it finishes it's work until T_{max}. Therefore, in the example depicted in figure 2, the L_I can be seen as the ratio of the area above the bar graph and below the line from $(0,T_{max})$ to (N_p-1,T_{max}), (which is also the area of the rectangle bounded by $(0,T_{avg})$ and (N_p-1,T_{max})), to the area of the entire rectangle bounded by $(0,0)$ and (N_p-1,T_{avg}) (T_{max}=89.92 in Figure 2). A high load balancing efficiency (~100%) indicates a program has good load balancing characteristics, i.e., fills most of the rectangle with useful work.

3 Implementation Example

As an example, consider the modified Sieve of Eratosthenes prime number program given in Appendix C, which finds all prime numbers between 1 and MAXN. The original program distributed a range of MAXN/N_p integers to each node, but the resulting load imbalance was severe. Figure 2 shows the load imbalance graphically for a 32 node run with MAXN=32,000,000 and a linear distribution of the integer domain *x* to each node (i.e. each node was assigned a range of 1,000,000 integers). By looking at the change in time between sample points (nodes), we can see that the amount of time required by node 0 for program completion is significantly smaller than the amount required for node 31 (22.2 seconds versus 89.9 seconds). Thus node 0 is idle 75% of the time. This indicates there is a large load imbalance here. Calculating the L_I and L_E using equations (2) and (3):

T_{avg}=2055.52/32=64.24

L_I=(89.92-64.24)/89.92=28.56%

L_E=100-L_I=71.44%

Since 28.6% of the run time is wasted with a linear distribution of domain *x*, it is reasonable to ask:

How can the range of integers be allocated to each node for the prime number search, such that each node will finish simultaneously?

An approximate function for the density of prime numbers in the range of integers [2,n] was given by Legendre as:

n/(ln(n)-1.08366); for n>2.

Inspection of the algorithm in Appendix C shows that the time spent in the main inner loop of the program is bounded by the square root of n, but only runs to the end of this loop when n is prime. Therefore the cumulative time function is expected to look like

$$t(x) = A \cdot \frac{x^{1.5}}{\log(x) - 1.08366}$$

where A is a machine dependent constant, and *x* is the upper bound on the range of integers for the program.

A curve fit of the last equation to actual program timings on a Cray T3D, gave the cumulative time function (see Figure 4):

$$t(x) = A \cdot \frac{x^{1.43}}{\log(x) - 1.08366} \qquad \textbf{(4)}$$

where A=3.9098E-7

A more accurate t(*x*) would not use a constant power (1.43) for *x*, but good results were obtained with this simplification since the power is almost constant.

Defining r=1.0/(ln(*x*) - 1.08366) to simplify notation, and taking the derivative with respect to *x* of **(4)**:

$$t'(x) = A \cdot r \cdot x^{0.43}(1.43 - r) \qquad \textbf{(5)}$$

Having an analytic formula for t'(*x*) allows us to use Newton's method more efficiently since we require only one ln(*x*) evaluation instead of 2 for each step of Newton's method. A was determined by solving equation (**4**) for A and then substituting known values for *x* and t(*x*) (determined from sample timings), as in

$$A = t(x)\left(\frac{\log(x) - 1.08366}{x^{1.43}}\right)$$

The range of numbers each node should work on can now be determined using two calls to Newton's Method to find x_i and x_{i+1}, where i is the node number, for 0 <= i <= N_p, with f(*x*) = t(*x*) - i * t(x_{max}), and
f'(*x*) = t'(*x*).

Here the theorem given in Appendix A can be utilized to guarantee Newton's Method will converge when the initial guess is in the interval (6, ∞). First x_L and x_U are determined from the theorem's given conditions. Writing out the equations (in addition to (4) and (5)):

f(*x*) = t(*x*) - i*TT/N_p
f'(*x*) = t'(*x*) = t(*x*)/*x**(1.43-r)
f''(*x*) = t''(*x*) = (t(*x*)*r/*x*+t'(*x*)*(.43-r))/*x*

We have discontinuities in all functions when ln(*x*)-1.08366=0 (ln(*x*)=1.08366, so *x*~=2.96).
Likewise, t'(*x*) is discontinuous at *x*=0, and t'(*x*) > 0 when 1.43 > r, or *x* > 5.94744. t''(*x*) > 0 when r+(1.43-r)*(.43-r) > 0, or r^2-.86*r+.61>0, which is positive whenever r is positive, i.e. when *x* > 2.96.
Therefore the conditions of the theorem are met when *x* > 5.94744, thus it is safe to set x_L = 5.95, and x_U = ∞ , to be the bounds of the *x* interval where Newton's Method is guaranteed to converge.
For the purposes of this program, we only need to know that *x*>=5.95 to ensure Newton's Method will converge, and this is checked in the code. With MAXN in the millions or billions, as is normal, the smallest initial guess for Newton's Method (*x*=MAXN/N_p) should never come close to *x*=6.

Timing results for $x_{max}=2^{28}$ on 16 PEs of a Cray T3D are shown in Figure 5. In this run $T_{avg}=1494.0$ and $T_{max}=1507.9$, so that $L_I=0.93\%$, and $L_E=99.07\%$, i.e. this decomposition is 0.93% slower than a perfectly balanced decomposition. This inefficiency is caused by the deviation of the actual timings from the timings predicted by Equation (**4**). The overhead time required to do the domain decomposition (Newton's Method) was a maximum of 0.000265 seconds for all nodes, or 0.000018 % of the total program run-time. Hence most of the wasted program run-time (0.93%) is caused by the deviation of the curve fitted $t(x)$ from the actual $t(x)$.

4 Heterogeneous Nodes

If the work is distributed amongst heterogeneous nodes, a similar procedure can be utilized so long as the relative performance of the various nodes is known and is constant over the entire domain (i.e. the ratio of timings on the various heterogeneous nodes for a given range of the domain is constant). Given the following definitions, we can derive a formula for distributing work amongst two different processor types.

t_{int} = Time interval assigned to all nodes,
N_1 = Number of type 1 nodes,
N_2 = Number of type 2 nodes,
T_1 = Total Time contribution from type 1 nodes
= $N_1 * t_{int}$,
T_2 = Total Time contribution from type 2 nodes
= $N_2 * t_{int}$,
$N_{eff} = N_1 + N_2$,
R = Ratio of type 1 node timings to type 2 node timings over the same interval of *x*.
R is assumed to be constant.
TT = Total Time for a program to run on one type 1 node = $t(x_{max}) = t(x_{Np})$,
T_{all} = Total Time for a program run on both type 1 and type 2 nodes.

Given N_1, N_2, R, and TT as data then an equal amount of work (time t_{int}) is assigned to all $(N_1 + N_2)$ nodes. This distributes the work so that all nodes finish simultaneously, partitioning the t-axis into $(N_1 + N_2)$ intervals of size t_{int}.

$$T_{all} = T_1 + T_2 = N_1 * t_{int} + N_2 * t_{int} = (N_1 + N_2) * t_{int},$$

and writing out the expression for TT we can see that by the definition of R:

$$TT = T_1 + R*T_2 = N_1*t_{int} + R*N_2*t_{int}$$
$$= (N_1 + R*N_2)*t_{int}, \text{ or}$$
$$t_{int} = TT/(N_1 + N_2*R)$$

Now
$$T_{all} = (N_1 + N_2) * t_{int}$$
$$= (N_1 + N_2)/(N_1 + N_2*R) * TT$$

and equation (1) becomes,
with $t(x)$ as defined on type 1 nodes:

for type 1 nodes, $0 <= i <= N_1$:

$$t(x_i) = i * T_{all}/N_{eff}, \text{ or } t(x_i) - i * t_{int} = 0,$$

and for type 2 nodes, since $t(x)$ was defined on type 1 nodes, we must scale it by 1/R, i.e. for $N_1 < i <= N_{eff}$:

$$t(x_i)/R = t(x_{N1}) + i * T_{all}/N_{eff}, \text{ or,}$$
since $T_1 = t(x_{N1})$, $t_{int} = T_{all}/N_{eff}$,

$$t(x_i)/R - T_1 - i * t_{int} = 0.$$

For instance, suppose the work is distributed among 7 type 1 nodes and 4 type 2 nodes, and the type 1 nodes execute the program in 3 times the amount of time type 2 nodes would require. Figure 3 shows this example for the case that the program takes 100 seconds to run on 1 type 1 node, and then
$t_{int} = TT/(N_1+N_2*R) = 100/(7+4*3) = 5.26$ seconds
$T_{all} = (N_1+N_2) * t_{int} = (7+4)*5.26 = 57.89$ seconds
$T_1 = N_1 * t_{int} = 7*5.26 = 36.84$ seconds
$T_2 = T_{all} - T_1 = 57.89-36.84 = 21.05$ seconds

The speedup (relative to running the program on 1 type 1 node, assuming no communication costs) on 11 type 1 nodes would be 11 fold. With the 4 faster type 2 nodes, the speedup is 100/5.26 = 19.0. In general, the speedup is N_1+N_2*R.

Generalizing to M types of nodes, when the following data is given:

$N_1, N_2, ... , N_M$,
R1(=1), R2, ..., R_M, (all time ratios are relative to type 1 nodes), and
$TT = t(x_{Neff})$
$= t(x_{max})$
= Total Time for a program run on one type 1 node.

Then
$N_{eff} = N_1+...+N_j+...+N_M$
speedup = $(N_1+N_2*R_2+...+N_j*R_j+...+N_M*R_M)$
(Relative to 1 type 1 node run)
t_{int} = TT/speedup
$T_{all} = N_{eff} * t_{int}$
$T_j = N_j * t_{int}$

$NSUM_j = N_1+...+N_j$, $NSUM_0 = -1$
$TSUM_j = T_1+...+T_j$, $TSUM_0 = 0$

and for type j nodes, $NSUM_{j-1} < i <= NSUM_j$:

$t(x_i)/R_j = TSUM_{j-1} + i * T_{all}/N_{eff}$, or
$t(x_i)/R_j - TSUM_{j-1} - i * t_{int} = 0$.

5 Multi-dimensional Domain

The following section outlines general conditions that must be satisfied by a load balanced program when the domain is multi-dimensional.

Definitions:

1) "Volume" is the data domain that is distributed over all of the processors in a partition of N_p nodes. This "Volume" is n-dimensional. The notation here attempts to remain consistent independent of the number of dimensions in V.
2) "Surface Area" is that part of a node's data domain that is adjacent to other nodes domains. The dimensionality is one less than that of the "volume".

Applying the method of sections III and IV to multiple dimensions is complicated by the fact that t(V) is a cumulative function. When generalizing to multiple dimensions, it is not known how to accumulate the time since this is application specific, i.e. depends upon how the data is distributed over the nodes. In HPF terminology, it depends upon whether the data is distributed in a cyclic, block, or degenerate fashion. Therefore this study concentrates on the sum-invariant part of the problem, which is defined as the *time density* (ρ), since it represents the amount of time contributed for each datum, i.e. for each volume element (voxel). Time density is the change in time for the change in volume represented by a given voxel (datum). Thus, the time density function can be defined similarly to other density functions, as a limit

$$\rho(X) = \lim_{\Delta V \to 0} (\Delta time)/(\Delta V)$$

Appropriate parts of the code can be timed to determine the amount of time required per datum as a function of the datum's placement (coordinates) in V. The timing data can be modeled by curve-fitting continuous real functions to discrete time samples, where these samples indicate the change in time for a given change in V. In this way a model of the time density function ρ(V) is obtained. In one dimension (i.e. in sections II-IV) ρ(V)=t'(*x*). The amount of time contributed per datum is known, and so the problem of partitioning the domain is reduced to determining how to sum the curve-fitted ρ(V) to get the appropriate sum (TT/N_p from Equation (**1**)) for a given node while ensuring all of the domain is distributed over V.

In general adding more voxels (more domain V) increases the time to program completion. If we restrict ourselves to nondecreasing t(V), i.e. $t(x_b)>=t(x_a)$ whenever $x_b>x_a$ for all dimensions of V, then ρ(V)>=0 for all V. Therefore, since a function which is non-negative and continuous (this is the "well-behaved" criteria mentioned earlier) on a given region S of V has a volume defined by:

,

$$t(V) = \int_S \rho(V)dV$$

it is seen that the integrals and the intervals can be arbitrarily subdivided without risk of cancellation since ρ(V) is nonnegative.

The time required for program completion for a given region S of V is the sum of the discrete time samples in interval S, which can be modeled as the sum of the curve-fitted ρ(V) over region S.

In general a load balanced program should satisfy:

$$TT = \int_{SS} \rho(V)dV$$
$$TT/NP = \int_{Si} \rho(V)dV$$
$$\bigcup_{\forall i} Si = SS$$

where
ρ(V) = Time Density function,
SS = all V in this program run,
TT = Total Time, the sum of ρ(V) over SS,
S_i = the region of V assigned to node i,
$0 <= i < N_p$.

In general for the multi-dimensional case, there is one equation, and M unknowns, where M is the number of dimensions in domain V. There are many application-specific considerations that must be taken into account when determining the best method for partitioning the data domain, both to minimize communication between nodes, and to ensure algorithmic efficiency. For example, it is often desirable to minimize the ratio of the "surface area" of a node's domain to the "volume" of the node's domain, in order to maximize the amount of computation done relative to the amount of communication done (this applies when the boundary "surface area" needs to be communicated). In this case, the ratio Area/Volume must be minimized for each S_i, where:

Ω_i is the boundary of S_i,

$$Area = \int_{\Omega i} dA$$
$$Volume = \int_{Si} dV$$

A complete description of data partitioning methods to reduce communication time and/or increase algorithmic efficiency is beyond the scope of this paper. However, these considerations produce other constraints upon the data distribution model, and constraints such as this can be used to reduce the data distribution problem to one variable (dimension) and one unknown.

By far the most common time density function is a constant amount of work per datum, as represented by ρ(V)=C, C>0, for arbitrary V. This can be called a linear (or homogeneous) distribution, and methods for evenly load balancing this distribution try to ensure each node gets an equal "volume" of V. In our generalization the "volume" does not have to be equal, but the change in time (sum of ρ(V)) assigned to a node should be equal. In the linear distribution case the change in t(V) is constant (see Figure 6) for a constant volume, so when the "volume" assigned to each node is constant, the time is also constant, and therefore equal.

To clarify the notation consider a fictitious multi-dimensional example. In a program with a 2- dimensional domain in which ρ(V)=C*(x+y), N_p=4, where (x,y) represent the domain that is being distributed over the nodes. Let x_{max}=y_{max}=20. Now in order to reduce the degrees of freedom we decide to break up the domain into rows, with each row containing work that will take TT/N_p seconds. Instead of the linear distribution seen in Figure 6, we see the distribution shown in Figure 7. First we find TT by summing ρ(V) over the entire domain V:

$$TT = \int_{SS} \rho(V) dV$$

$$TT = \int_0^{20}\int_0^{20} C \cdot (x+y) dx dy$$

$$TT = (C/2) \cdot (x^2 y + y^2 x)\Big|_0^{20}\Big|_0^{20}$$

so TT=C*8000.

Equating i*TT/N_p to the SUM of ρ(V) for a row bounded by x in (0,x_{max}) and y in (0,y):

$$\frac{TT}{NP} \cdot i = \int_0^{20}\int_0^{xmax} C \cdot (x+y) dx dy$$

$$\frac{TT}{NP} \cdot i = (C/2) \cdot (x^2 y + y^2 x)\Big|_0^{xmax}\Big|_0^{x2}$$

$$\frac{TT}{NP} \cdot i = (C/2) \cdot (x_{max}{}^2 y + y^2 x_{max})$$

Now x_{max} = 20, and TT = C*8000, hence

$$\frac{C \cdot 8000}{NP} \cdot i = C \cdot (200 \cdot y + 10 \cdot y^2)$$

or, cancelling C and rewriting,

$10*y^2 + 200*y - 8000*i/N_p = 0$.

Solving explicitly for y using the quadratic equation:

$$y_i = \frac{-200 \pm \sqrt{200^2 - 40 \cdot -8000 \cdot i/NP}}{2 \cdot 10}$$

Obviously negative solutions are not desired, hence

$$y_i = -10 + \sqrt{1 + 2 \cdot i}$$

So for i=0..4, y_i = {0, 7.32, 12.36, 16.46, 20}

The fractional remainders here indicate that we will not be perfectly load balanced, unless we are willing to assign partial rows to the processors. This can be termed the "perfection complex", since it demonstrates the perfection/complexity (fine/coarse-grained) trade-off, i.e. allowing partial rows would allow perfect load balance, but would increase program complexity.

6 Automation

The next possible application (or evolution) of this method would be towards automated load balancing, i.e. attempting to eliminate load imbalance as a significant threat to scalability through the use of an automated parallel programming tool which incorporates this method in it's suite of options. Related work can be found in [1]. A system can be envisioned wherein HPF data distribution directives are used in combination with this tool to develop a time density profile for a given code. The first pass through the code determines how much time is spent to do calculations on a given voxel in the code's data domain (arrays). A time density (ρ) array is thus generated with a duplicate distribution over the processors as the original array(s). By having a duplicate distribution for the ρ array, we are ensured that this processor's references to the ρ array are local to this processor. This ρ array can then be utilized to perform the load balance algorithm described in Section II of this paper to dynamically refine the distribution of the array associated with this ρ array over the processors. Although in general Newton's Method could not be used (since an analytic t'(V) is not available, but discrete time samples are available), the secant method could be used. However, the impact of this ρ array on memory usage would be quite dramatic, since memory usage would be doubled for distributed arrays. This problem could be alleviated by combining elements of the time density array, e.g. by adding groups of 5 elements along one dimension of the array to reduce ρ array memory usage (and timing overhead) by a factor of 5, but there is an obvious memory to load balance efficiency (fine/course grained) trade-off here. The tool would have to provide the interface to the programmer to allow them to define: which arrays will be timed, where a pass starts and stops (generally a synchronization point), and how (and if) a given ρ array should be "compressed".

Also, if the computational load for individual voxels changes over time, it may be required to periodically check whether the program has drifted out of balance. For instance, by checking whether the maximum and minimum node times are more than a threshold value apart at every 10th pass (timestep), e.g. if $(T_{max}-T_{min})/T_{avg} > 10\%$, then it's time to dynamically load balance the code, i.e. to go through the above domain distribution procedure again. However, checking this threshold value requires a global summation, so it cannot be checked often without incurring excessive overhead. Likewise, the redistribution generally requires global communication, and so the threshold value cannot be set to be too low without incurring excessive overhead from excessive numbers of redistributions. The parallel programming tool would have to allow the number of passes per check and the threshold value to be supplied by the programmer, to allow them to affect this overhead/load balancing efficiency trade-off.

7 Conclusion

A load balancing method for parallel processing programs has been developed that allows programs that have a computational load which varies as a function of the data domain to be load balanced. This method has been generalized to run on heterogeneous as well as homogeneous processors (with certain assumptions). The method was applied to a problem with one degree of freedom: the generation of prime numbers in the field of reals with a modified Sieve of Eratosthenes algorithm. This method increased the Load Balance Efficiency of this program running on 32 nodes from 71% to over 99%. The same method has also been generalized to multiple dimensions.

Although any root finding method can be employed in this method, the superior convergence properties of Newton's method strongly favors its use in the implementation of this method, since the root finding is overhead for the program.

Incorporating this "functional" load balancing method into a parallelization tool may allow it to automatically load balance certain applications across any number of different types of nodes.

8 Acknowledgments

This work was conducted at the U.S. Environmental Protection Agency's National Environmental Supercomputer Center in Bay City, MI.

The author thanks Dr. George Delic and Chris Sarnowski for their comments and suggestions.

9 References

[1] Tomko, Karen A., *Domain Decomposition, Irregular Applications, and Parallel Computers*, 1995, University of Michigan.

[2] Burden, R.L., and Faires, J.D. 1988. *Numerical Analysis,* fourth edition. Boston, Massachusetts: PWS-Kent Publishing, pp. 51.

[3] Demidovich, B.P., and Maron, I.A. 1987. *Computational Mathematics.* Moscow, Russia: MIR Publishers, pp. 128.

APPENDIX A : Conditions for the Convergence of Newton's Method

From the programmer's point of view, an interval of guaranteed convergence for Newton's Method would allow the method to be used without concern about the method diverging. Also, the programmer is attempting to minimize the overhead associated with this load balancing method, so a quickly converging method is strongly preferred over slower methods with poorer convergence properties. The following theorem can be applied to determine the interval on which Newton's Method will converge. For further discussion see [2]:

Let $f \varepsilon C^2[a,b]$. If $p \varepsilon [a,b]$ is such that $f(p)=0$ and $f'(p)\neq 0$, then there exists $\delta>0$ such that Newton's Method generates a sequence $\{p_n\}$ for $n=1,.., \infty$ converging to p for any initial approximation
$p_0 \varepsilon [p-\delta,p+\delta]$.

When Newton's method is viewed as a contraction mapping, it is found that the function

$$Q(x) = \frac{f(x)\cdot f''(x)}{f'(x)^2}$$

determines the rate of convergence. Thus, for any interval of x where $Q(x) < 1$, Newton's Method will converge to the root for any choice of an initial guess within this interval.

Alternatively, for the purposes of this discussion, it is known that $t(x)$ is non-decreasing, so $f(x)=t(x)-C$ is also non-decreasing. In general $t'(x) > 0$, since every voxel requires <u>some</u> computation, so $t(x)$ is strictly increasing. Hence the following theorem defines a result relevant to this method (see [3]):

Let $f \varepsilon C^2[x_L,x_U]$. If $x_R \varepsilon [x_L,x_U]$ is such that $f(x_R)=0$, (so $f(x_L)f(x_U)<0$), and for all $x \varepsilon [x_L,x_U]$, $f'(x)$ and $f''(x)$ are nonzero and preserve signs, then Newton's method generates a sequence $\{x_i\}$ for $i=0,.., \infty$, converging to x_R for any initial approximation $x_0 \varepsilon [x_L,x_U]$, so long as $f(x_0)f''(x_0) > 0$.

In fact the $f(x_0)f''(x_0) > 0$ condition can be removed, so long as $f'(x)>0$ and $f''(x)>0$ over all $x \varepsilon [x_L,x_U]$.
Applications are not guaranteed to have $f''(x) > 0$ (since this is application dependent), but in general $f'(x) > 0$ when applying this method.

APPENDIX B : Modified Sieve of Eratosthenes Prime Number algorithm using the load balancing method

To find all primes in 1..MAXN, for node i in 0..N_p-1, implement the following algorithm:

1. All nodes find_primes in 2..SQRT(MAXN), save in prime_low.
2. Call NEWTON(x_i) to find the lower bound of

integers assigned to each node.

3. Call NEWTON(x_{i+1}) to find the upper bound of integers assigned to each node.
4. All nodes find_primes in $x_i..x_{i+1}$
5. Output timings and prime numbers.

Pseudocode for Find_primes in step 1 and 4 is as follows:

```
DO n=NLO, NHI, 2
  DO ip=3,mxp
    notprime = (mod(n,prime_low(ip))=0)
    if (notprime) goto found
  enddo
 found:
  if (not notprime) then
    nbrprimes = nbrprimes + 1
    prime(nbrprimes) = n
  endif
  mxp = maximum prime index
enddo
```

Pseudocode for the NEWTON method in steps 2 and 3 is as follows:

```
Call FUNCD(MAXN,TT,Dummy)
TT_PE = TT * i/$N_p$ the portion of work allocated up to this PE.
$x$   = MAXN * i/$N_p$ (initial guess is the linear distribution).
Check whether $x0$ is within bounds
DO J=1,100
  Call FUNCD($x$,F,DrF)
  DX = (F-TT_PE)/DrF
  $x$=$x$-DX
  if ((-RTNEWT)*(RTNEWT-MAXN)<0) then
                 print Jumped out of bounds message
                 exit
                 endif
  if (ABS(DX).LT.EPSILON) then
                 NEWTON=$x$
                 return
                 endif
enddo
print NEWTON exceeded maximum number of iterations
end
```

Pseudocode for the FUNCD call above follows, implementing the function f(x) and f'(x) defined in Section IV:

```
P=1.43
r = 1.0/(log($x$)-1.08366)
$xp = x^P$
FUNC=A*$x$p*r
Deriv = FUNC*(P-r)/$x$
end
```

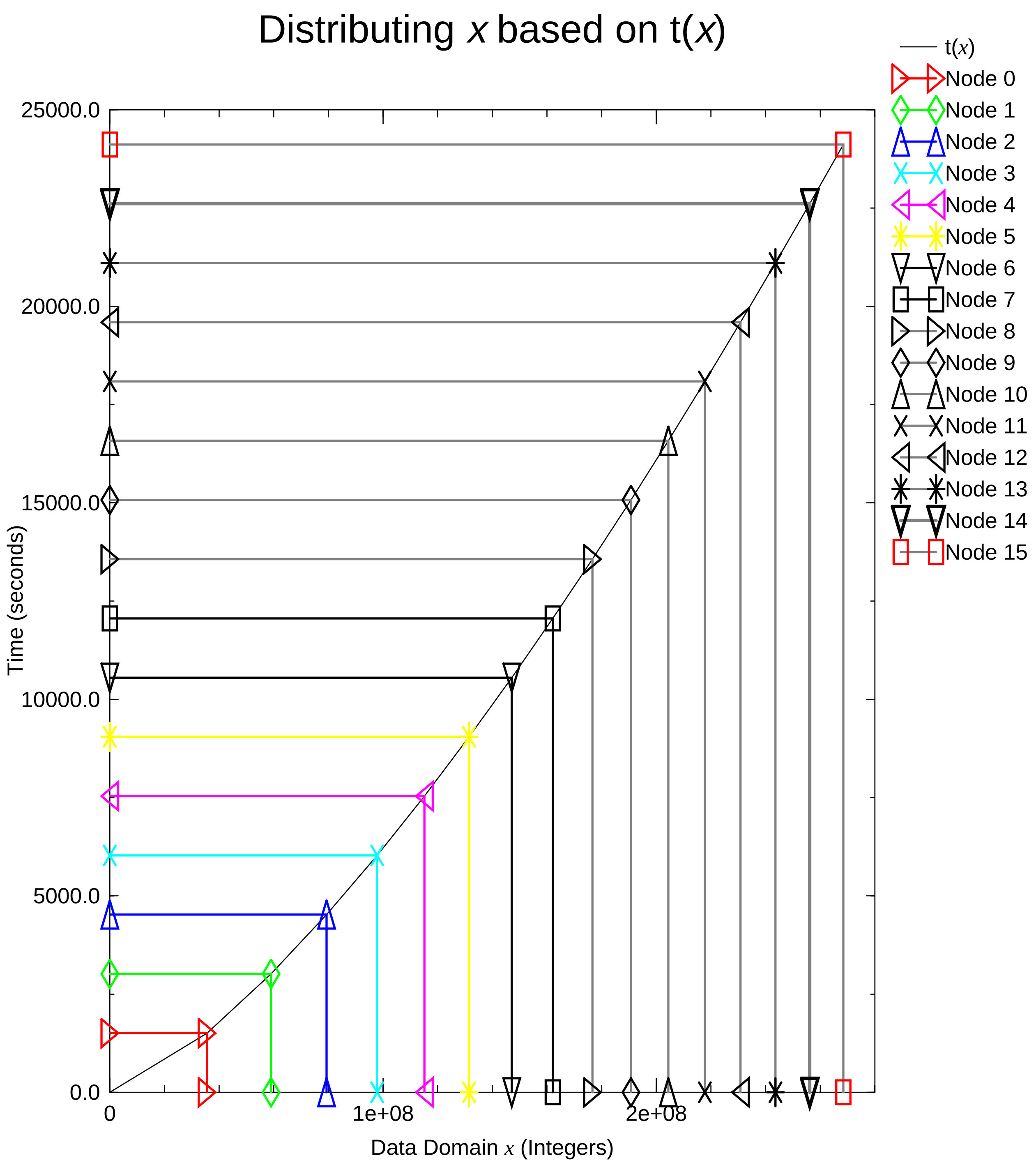

Figure 1 depicts how the time axis is subdivided into equally spaced intervals for an arbitrary t(x) function. This particular t(x) is taken from the application described in Section IV, i.e. equation (4).

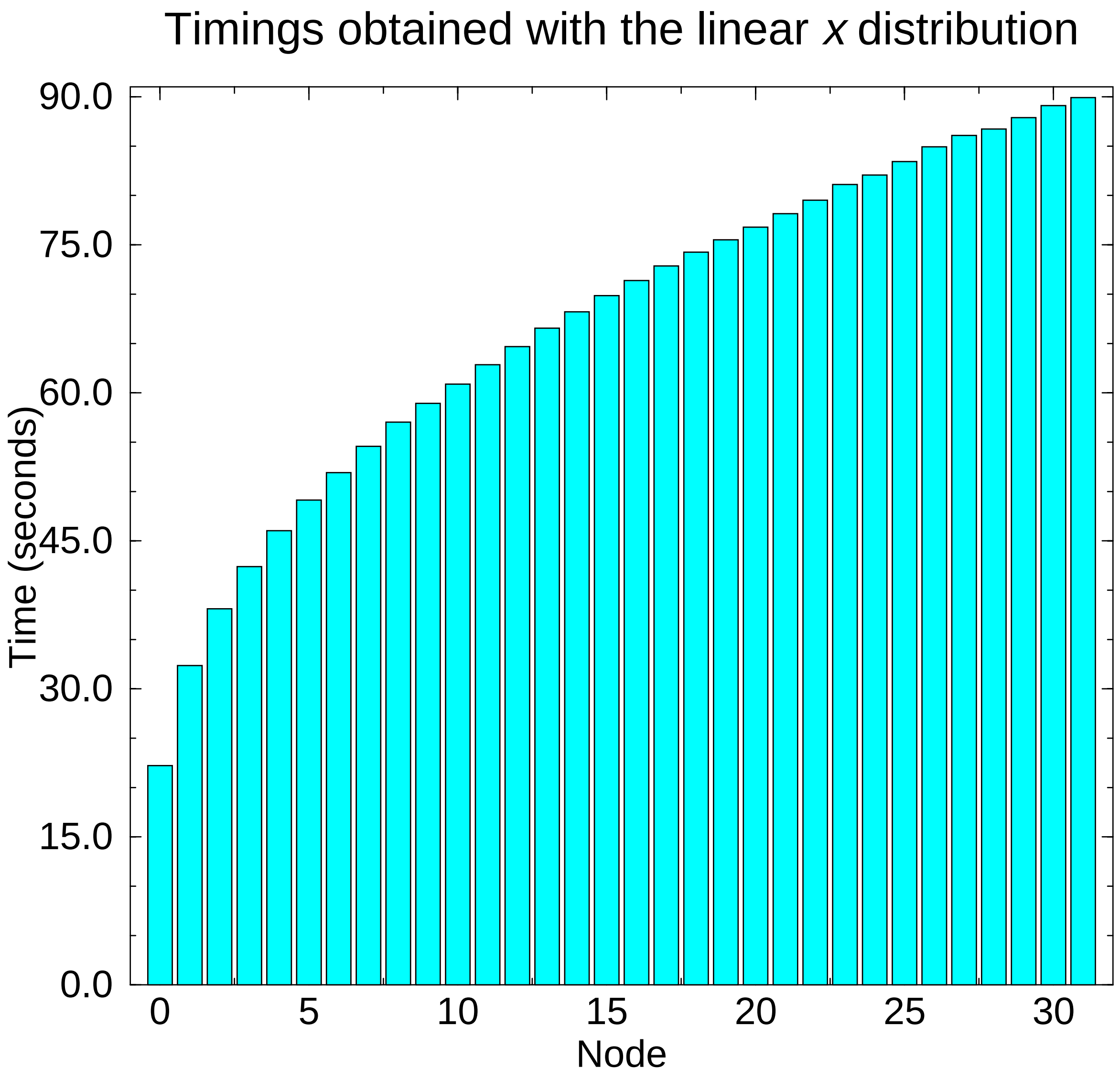

Figure 2: Actual timings of the Prime number program described in section IV when *x* is distributed using the linear distribution, wherein each node is given a range of integers of equal length. The wide variance in time-to-completion for this case motivated both the development of this load balancing method and the definition of the L_E and L_I measures of load balance in section II.

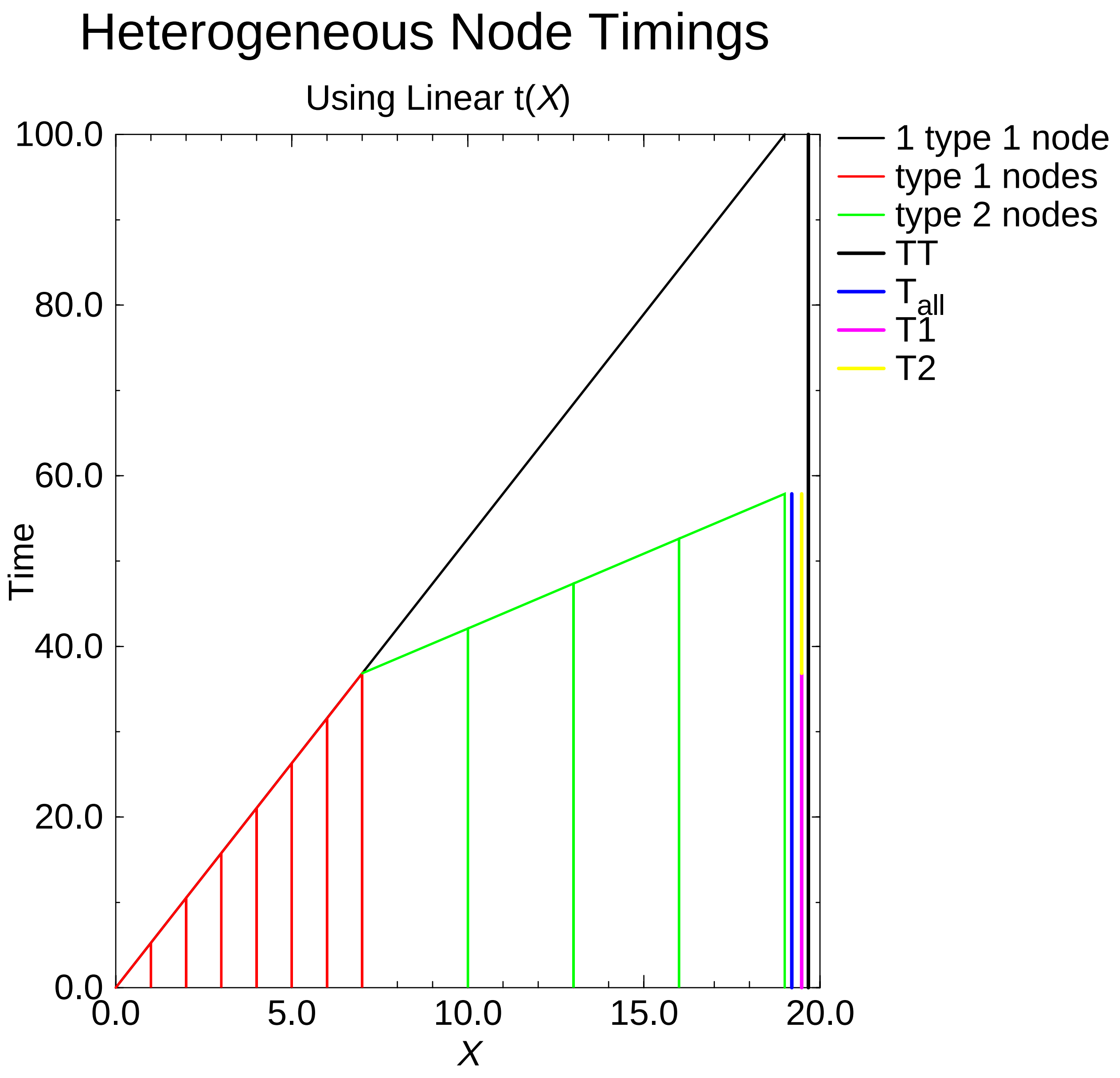

Figure 3 depicts the heterogeneous node example for a linear function t(*x*) with the two node types described in Section III.

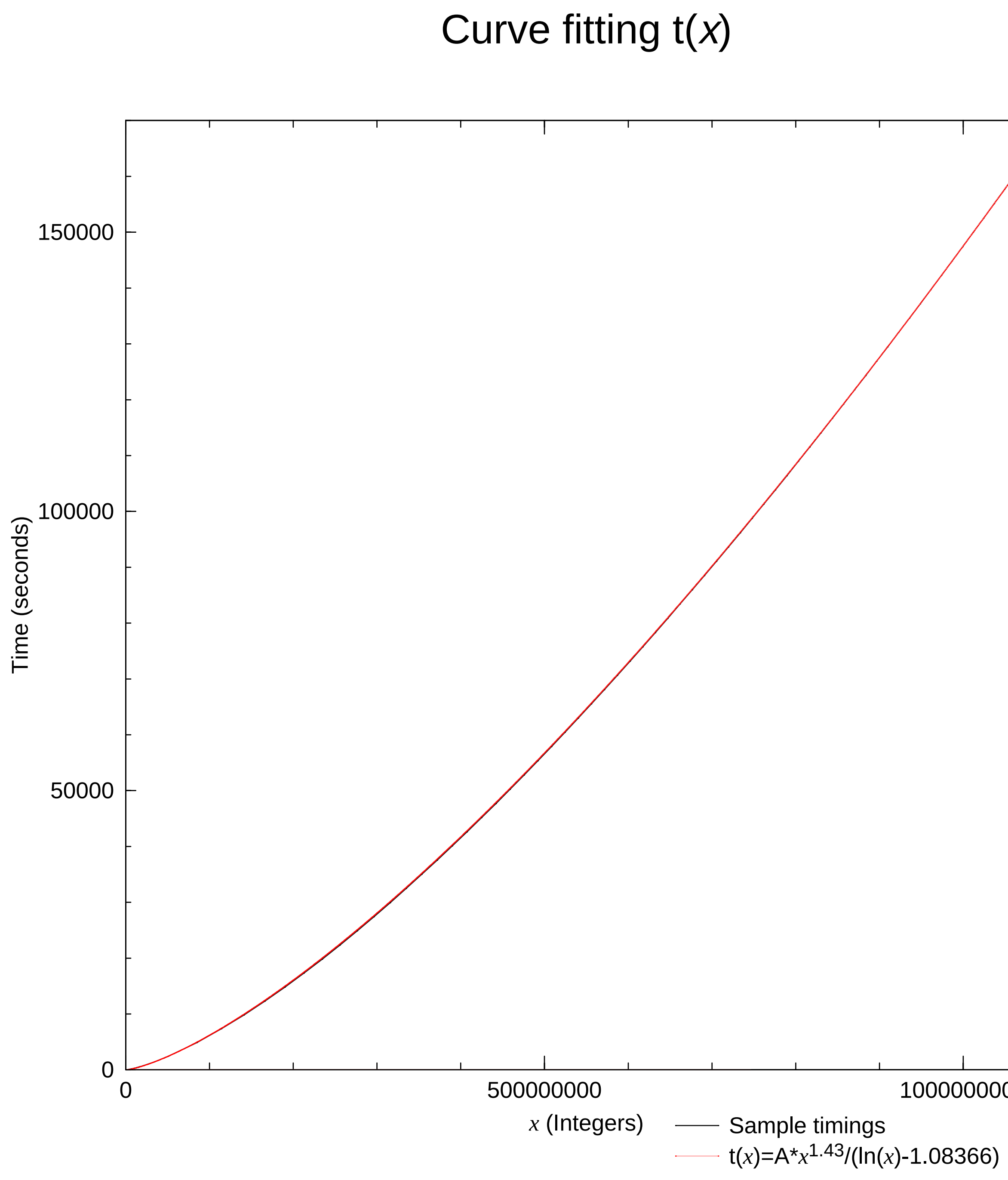

Figure 4: Comparison of fitted and empirical results for the function t(x) of Equation (4) when both are superposed.

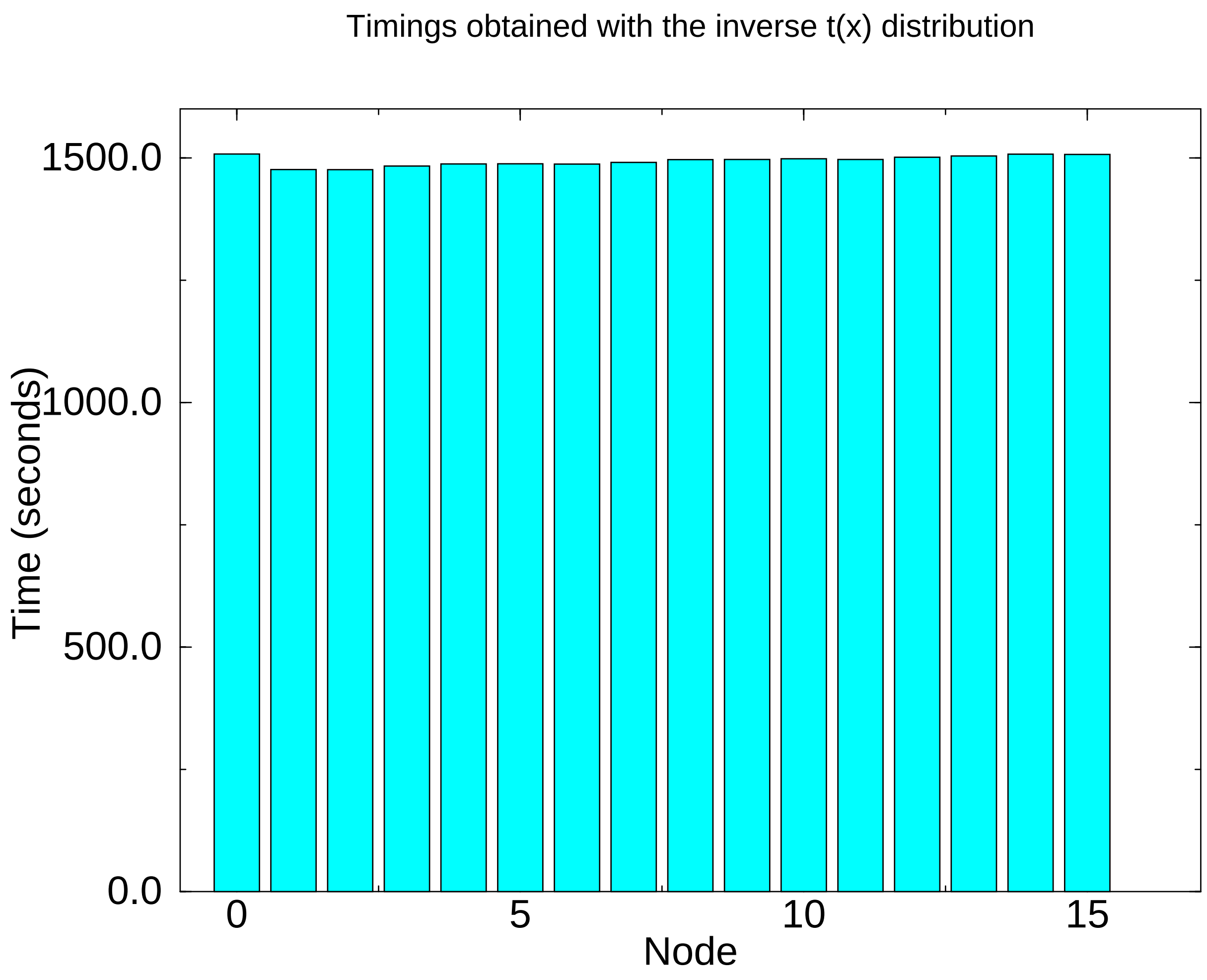

Figure 5 shows the timing results for MAXN=x_{max}=2^{28} on 16 PEs of a Cray T3D.

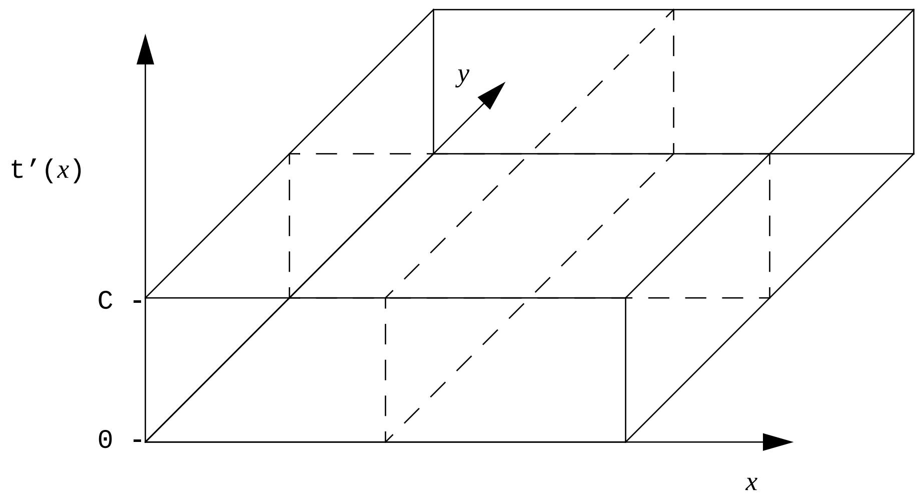

Figure 6 shows one possible data partitioning for a 2 dimensional x distributed across 4 nodes when $\rho(x)$=C.

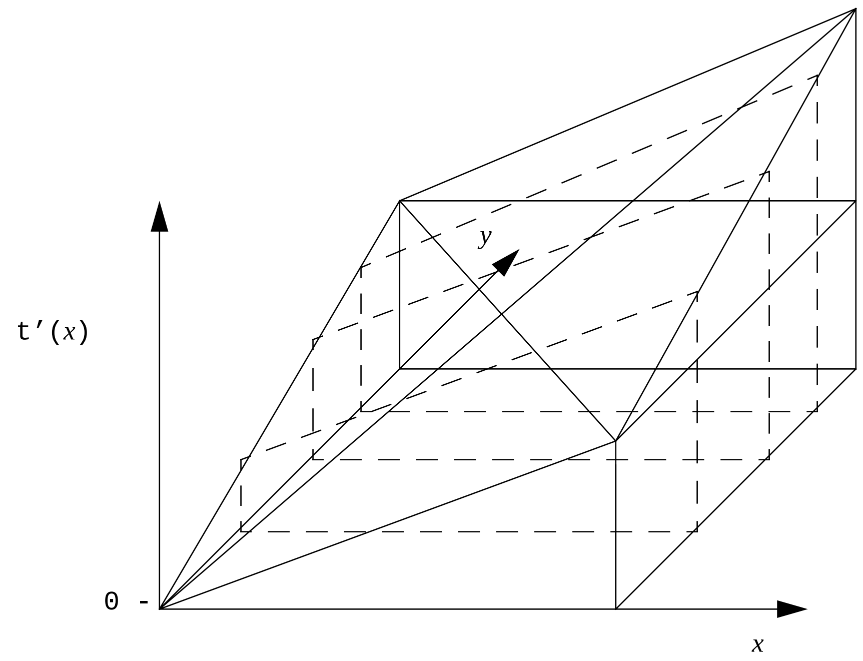

Figure 7 shows one possible data partitioning for a 2 dimensional *x* distributed across 4 nodes when ρ(*x*)=C*(*x*+*y*).

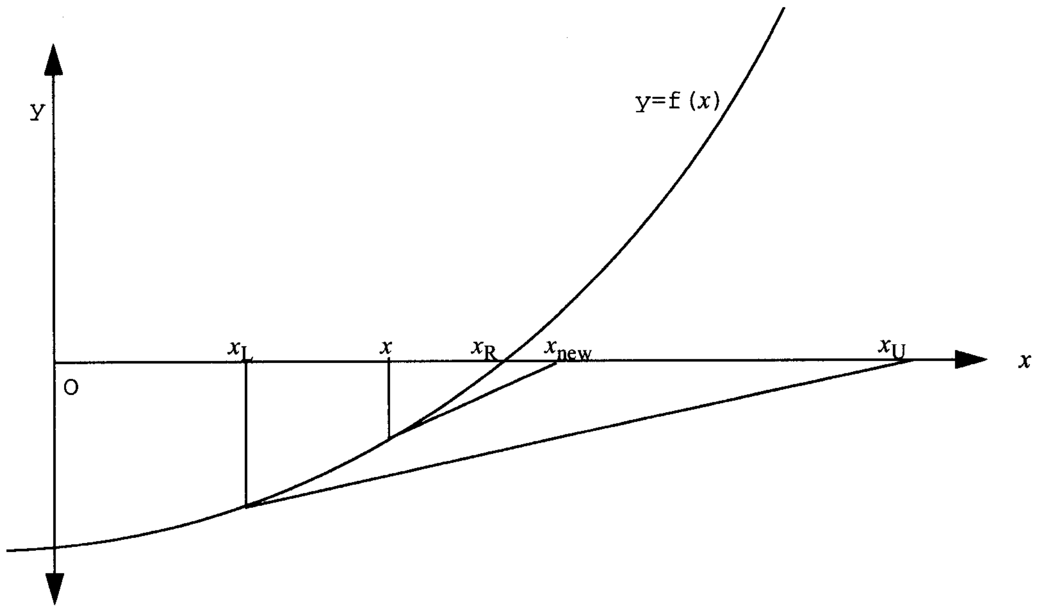

Figure 8 depicts an interval [x_L,X_U] with an f(x) satisfying the conditions of Swarz's theorem.