Interval Computations
No 4, 1993

Estimating Errors of Indirect Measurement on Realistic Parallel Machines: Routings on 2-D and 3-D Meshes That are Nearly Optimal

Elsa Villa, Andrew Bernat, and Vladik Kreinovich*

Decreasing of running time while estimating the accuracy of indirectly estimated function by mean of several parallelized processors with different parallelization schemes is considered. In particular, we describe routing methods that will enable us to implement these fast parallel algorithms on 2-D and 3-D meshes in a nearly optimal manner.

Оценка ошибок косвенного измерения на существующих параллельных компьютерах: почти оптимальные маршрутизации на двух- и трехмерных сетях

Э. Вилла, Э. Бернат, В. Крейнович

В статье рассматривается снижение времени вычисления для оценки погрешности косвенно вычисленной функции с применением нескольких параллельно работающих процессоров при использовании различных схем параллелизации. В частности, описываются методы маршрутизации, которые позволят нам применять быстрые параллельные алгоритмы на двух- и трехмерных сетях почти оптимальным образом.

*This work was supported in part by NSF Grant No. CDA-9015006 and by a grant from the Institute for Materials and Manufacturing Management. The authors are greatly thankful to Misha Koshelev for his help with the figures, and to anonymous referees for important suggestions that helped to improve the algorithms and to clarify the exposition.

1 Introduction: estimating errors of indirect measurements is one of the main applications of interval computations

Indirect measurements are a frequent engineering problem. In many real-life problems, we are interested in the value of some physical quantity y that is difficult or impossible to measure directly. For example, we may be interested in the amount of oil in a given area. In such cases, to estimate y, we measure other parameters $x_1, \ldots, x_n$ that are somehow related to y, and then reconstruct y from the results $\tilde{x}_i$ of these measurements. Such a situation is called an *indirect measurement* of y.

An algorithm f that transforms the results $\tilde{x}_i$ into an estimate $\tilde{y}$ for y is often extremely complicated; e.g. in geophysics, this algorithm is actually a numerical method for solving a complicated system of non-linear partial differential and/or integral equations describing how electromagnetic and ultrasound waves travel inside different layers.

These algorithms are even more complicated in intelligent systems, when to process the results of direct measurements $\tilde{x}_i$, we use an inference engine of an expert system or a neural network.

It is necessary to estimate the errors of indirect measurements. Indirect measurement leads to a value $\tilde{y}$. What is the accuracy of this value? In many applications, this is a crucial question. Suppose, e.g., that the estimated amount of oil in some area is 100 million tons. If the accuracy of this measurement is ± 10, this means that there sure is sufficient oil in this area to make drilling economically attractive. However, if the accuracy is ± 100, then it is quite possible that there is no oil at all, and it would be better to perform more accurate measurements before making such an investment.

The error in y comes from the errors in $\tilde{x}_i$. These values $\tilde{x}_i$ are measured by regular measuring devices, for which the manufacturers usually supply the accuracy information. So, we have the following problem: we know the accuracy of $\tilde{x}_i$; what is the resulting accuracy of the indirect measurement?

In some cases, these estimates are known from numerical mathematics, but in many cases, such estimates are not known. For some numerical algorithms f, accuracy estimates are known from numerical mathematics. However, for other situations no ready-to-use estimates exist.

In order to describe how these estimates are now calculated, let us recall in what form the accuracy estimates for measuring x_i are usually provided by the manufacturer of the measuring device.

How errors are described. In some cases, the only thing that the manufacturer guarantees is that the error does not exceed a certain value Δ_i. In other words, if the measured value equals $\tilde{x}_i$, then the actual value x_i of this physical quantity belongs to an interval $[\tilde{x}_i - \Delta_i, \tilde{x}_i + \Delta_i]$. In this case, we are given:

- an algorithm f that transforms n real numbers $x_1, \ldots, x_n$ into a value $f(x_1, \ldots, x_n)$;
- n real numbers $\tilde{x}_i$, $1 \le i \le n$;
- n positive real numbers Δ_i, $1 \le i \le n$,

and we need to compute the interval of possible values of $f(x_1, \ldots, x_n)$ when $x_i \in [\tilde{x}_i - \Delta_i, \tilde{x}_i + \Delta_i]$.

Another typical situation is when the measuring device has been *calibrated*, i.e., the results of measurements have been compared with the results obtained by some more accurate measuring device. This comparison provides us not only with the largest possible value Δ_i of the error, but also tells how frequent different values of error are. In other words, it provides us with some information about the statistical characteristics of the error. Usually, these statistical characteristics are the average error and the standard deviation.

As soon as we know the average error of this measuring device we can then, subtract this value (the *systematic error*), and thus eliminate this component of an error. Therefore, in cases when calibration was applied, the errors $\tilde{x}_i - x_i$ are a random variables with 0 average and known standard deviation σ_i.

How errors of indirect measurements are estimated now. There are three main methods of error estimation: interval computations, methods based upon numerical differentiation, and Monte-Carlo methods.

Interval computations. The main idea of interval computations (see, e.g., [6]) is as follows: the algorithm f, no matter how complicated it is, consists of elementary computational steps ($+, -, \times$, etc). We know only

the interval of possible values of the inputs. Therefore, after each step, we obtain an interval of possible values of the corresponding intermediate results. So, on each step, instead of applying the corresponding arithmetic operation to numbers, we apply it to intervals. At the end, we obtain an interval that contains all possible values of y.

This is one of the main applications of interval computations to real-life problems.

The main drawback of the traditional interval computations approach. If we apply this idea "mechanically", by just changing all operations to operations with intervals, then the resulting estimate is often an "overshoot" (see, e.g., [6]). There exist many methods of "restructuring" the algorithm that enable us to avoid such an overshoot, but again, these methods are very specific, and they have been developed only for certain classes of algorithms that in no way exhaust the list of all possible algorithms.

Alternative method of computing the interval of possible values of $f(x_1, \ldots, x_n)$: numerical differentiation. Other methods that give exactly the interval of possible values of $f(x_1, \ldots, x_n)$ (and not a bigger interval) are based on the assumption (usually true) that the measurements are so accurate, that we can neglect the terms that are quadratic in errors. For example, if the measurements are done accuracy 2% (0.02), then the quadratic terms are proportional to 0.0004 (0.04%).

If we denote the errors by $\Delta y = \tilde{y} - y$ and $\Delta x_i = \tilde{x}_i - x_i$, we can then conclude that

$$\Delta y = \tilde{y} - y = f(\tilde{x}_1, \ldots, \tilde{x}_n) - f(x_1, \ldots, x_n) = f(\tilde{x}_1, \ldots, \tilde{x}_n) - f(\tilde{x}_1 - \Delta x_1, \ldots, \tilde{x}_n - \Delta x_n).$$

If we expand the right-hand side into a Taylor series and neglect quadratic terms, we conclude that $\Delta y = f_{,1}\Delta x_1 + \cdots + f_{,n}\Delta x_n$, where $f_{,i}$ denotes the partial derivative $\partial f/\partial x_i$.

Therefore, if we know the values Δ_i such that $|\Delta x_i| \le \Delta_i$, then the possible values of Δy form an interval $[-\Delta, \Delta]$, where $\Delta = |f_{,1}|\Delta_1 + \cdots + |f_{,n}|\Delta_n$.

Since we are considering a complicated case, when an algorithm f is not simply an explicit expression, but a very complicated algorithm, it is impossible to differentiate f analytically. To obtain numerical estimates, we use the same assumption that the terms that are quadratic in errors are negligible. In this case, for each i, if we take h small, we will conclude that

$f(\tilde{x}_1, \ldots, \tilde{x}_{i-1}, \tilde{x}_i + h, \tilde{x}_{i+1}, \ldots, \tilde{x}_n) = f(\tilde{x}_1, \ldots, \tilde{x}_i, \ldots) + hf_{,i}$. Therefore, we estimate $f_{,i}$ as $\big(f(\tilde{x}_1, \ldots, \tilde{x}_{i-1}, \tilde{x}_i + h, \tilde{x}_{i+1}, \ldots, \tilde{x}_n) - \tilde{y}\big)/h$. This formula is the simplest numerical algorithm for computing $f_{,i}$ and therefore, the entire method is called *numerical differentiation.* According to this method, we apply the algorithm f $n+1$ times: first, to compute $\tilde{y}$, and then n more times, to estimate the partial derivatives. Then we compute Δ.

Main drawback of numerical differentiation. This method computes the exact interval of possible values (not taking into consideration negligible quadratic terms, of course), but it requires that we compute f $n+1$ times. For many real-life problems (e.g., for the analysis of a geophysical data), the number of inouts n can be in thousands, and each computation of f is (already) very time-consuming. As a result, computing Δ takes *too much time.*

Monte-Carlo method for estimating an interval. A faster method that was proposed in [5] is termed the *Monte-Carlo* method. According to this method, we *simulate* errors, i.e., use a computer random number generator to generate random numbers ξ_i that are distributed according to Cauchy distribution with a density $\rho(x) = \text{const}/(1 + (x/\Delta_i)^2)$ with 0 average and (scale) parameter Δ_i. Then, we compute $\Delta y^{(1)} = \tilde{y} - y^{(1)}$, where $y^{(1)} = f(\tilde{x}_1 - \xi_1, \ldots, \tilde{x}_n - \xi_n)$. In the case that we may neglect terms that are quadratic in error, we can conclude that $\Delta y^{(1)}$ is a Cauchy-distributed random variable with 0 average and parameter $\Delta = \sum |f_{,i}|\Delta_i$. So, to determine Δ, we repeat this procedure several times, obtaining N values $\Delta y^{(1)} = \tilde{y} - y^{(1)}$, $\ldots$, $\Delta y^{(N)} = \tilde{y} - y^{(N)}$, and then apply standard statistical techniques (namely, Maximum Likelihood Method *MLM*) to estimate Δ. For Cauchy distribution, MLM turns into solving an equation $\sum 1/\big(1 + (y^{(k)}/\Delta)^2\big) = N/2$ (an algorithm for solving this equation is given in [5]). For $N = 50$, we get Δ with a 20% accuracy in $\geq 99.9\%$ of cases. A 20% accuracy is quite sufficient if we take into consideration that this is a precision with which we know accuracy. There is little difference between a measuring device with a 2% accuracy and a device with a 2.1% accuracy.

This method executes f $N + 1 = 51$ times. So, for large n, this method is much faster than a numerical differentiation method.

What is the advantage of using the above-described statistical method for estimating an interval instead of more traditional interval methods? The main advantage is this: as a result of traditional interval methods, we often

get an overestimate of the desired interval

$$f\big([\tilde{x}_1 - \Delta_1, \tilde{x}_1 + \Delta_1], \ldots, [\tilde{x}_n - \Delta_n, \tilde{x}_n + \Delta_n]\big).$$

This overestimate can be large. On the other hand, the above-described statistical method gives an interval that (with a 99.9% guarantee) differs from the desired interval by $\leq 20\%$.

Similar methods can be used to compute the statistical characteristics of the random error of an indirect measurement. If all the errors Δx_i are independent random variables, with 0 average and standard deviations σ_i, then Δy is also a random variable with 0 average and standard deviation $\sigma = \sqrt{f_{,1}^2\sigma_1^2 + \cdots + f_{,n}^2\sigma_n^2}$. As soon as we know partial derivatives (e.g., by applying numerical differentiation), we can use this simple formula to compute σ.

Another possibility is to use a *Monte-Carlo* method. According to this method, we *simulate* random errors, i.e., use a computer random number generator to generate random numbers ξ_i that are distributed with 0 average and standard deviation σ_i. Then, we compute $\Delta y^{(1)} = \tilde{y} - y^{(1)}$, where $y^{(1)} = f(\tilde{x}_1 - \xi_1, \ldots, \tilde{x}_n - \xi_n)$. In the case that we may neglect terms that are quadratic in error, we easily conclude that $\Delta y^{(1)}$ is a random variable with 0 average and standard deviation σ. So, to determine σ, we repeat this procedure several times, obtaining N values $\Delta y^{(1)} = \tilde{y} - y^{(1)}$, $\ldots$, $\Delta y^{(N)} = \tilde{y} - y^{(N)}$, and then apply standard statistical techniques to estimate σ, e.g.,

$$\sigma \approx \sqrt{\frac{(\Delta y^{(1)})^2 + \cdots + (\Delta y^{(N)})^2}{N}}.$$

For $N = 25$, we get σ with a 20% accuracy in $\geq 99.9\%$ cases (We have already argued that such an accuracy is quite sufficient if we take into consideration that this is a precision with which we know accuracy).

This method executes f $N + 1 = 26$ times.

These methods may be easily parallelized. In all these methods, the most time-consuming part of the algorithm is applying a time-consuming algorithm f to different data. So, a natural idea to save time is to make all these calls of f handled by separate processors. Then, if we have enough processors (i.e., at least one for each call of f), the resulting computation time will be equal to the time necessary to apply f only once.

As a result, if we have several processors working in parallel, then we

may compute both the estimate $\tilde{y}$ and its accuracy in practically the same time that we would have spent on an estimate $\tilde{y}$ itself.

Communications necessary for the resulting parallel algorithm. We will assume that a program that computes f is already installed in all the processors. We must allocate one of the processors as the one through which the system will communicate with the outside world: i.e., the one that obtains the measured values $\tilde{x}_i$ and their accuracy characteristics Δ_i (or σ_i), and that returns the resulting accuracy estimate Δ (or σ) to the user. We will call this processor a *Central Processor* (CP for short).

Before each processor starts computing, it must receive from CP the values $\tilde{x}_i$ and the accuracy characteristics Δ_i (or σ_i). After each processor is done, it must send the results of its computations back to CP. Based on these results, the CP will estimate Δ or σ.

During the computations, no communications are necessary.

This algorithm has been successfully implemented on a special architecture. The majority of the existing concurrent systems are designed to handle algorithms with extensive communication, so they usually have a very complicated communication protocol designed to resolve write-read conflicts, to make sure that both communicators are ready, etc. As a result, each communication takes considerable time.

For our particular problem, we do not have any conflicts to resolve, we only need to pass the information before the computations and to collect the results after it. So, for this purpose, a simplified concurrency scheme will be quite sufficient (and even preferable, since it does not waste time on unnecessary checks). Such a system (called BaRe, a short for *B*roadcast *a*nd *Re*plication) was used at the University of Texas at El Paso in 1989–91 (for details, see [7]).

Using this system, we implemented the above algorithms and showed that we really obtained the desired speed-up [5].

In this paper, we will provide the description of routings for realistic computer architectures.

2 Estimating errors of indirect measurements on a two-dimensional mesh

2.1 Why 2-D mesh?

The existing hardware technology mainly supports 2-D designs. The existing hardware chip technology is mainly oriented towards a planar design, so we end up with processors densely packed on a plane. Such a configuration, in which processors are arranged into an $m \times m$ grid such that each processor is directly connected with its four neighbors, is called a *two-dimensional mesh* (see, e.g., [1, 2]).

Where to place a central processor (CP)? In a 2-D mesh, all processors are considered equal in quality, so in principle, we can use any of them as a central processor (that does all the communications with the user).

After the central processor receives the user's problem, it has to communicate the problem and the data associated with it to every other processor in the network. Communication is time consuming; therefore, we will place this communicating processor *in the center of the mesh* so that the distance to the farthest processor is minimized.

How many processors to use? As we have described, there are two basic ideas: numerical differentiation and Monte-Carlo methods. Numerical differentiation requires $n + 1$ calls of f (so, $n + 1$ processors, where n is the total number of variables). Monte-Carlo methods require correspondingly 26 and 51 processors (26 for σ, and 51 for Δ).

If the total number of measurement is small ($n < 26$), then it is better to use the methods of numerical differentiation. However, when n is small, estimating f is not a large problem, and it may not be worthwhile to use parallel computations.

The problem becomes really serious and time-consuming when we have large numbers of measurement results (n up to 10^4 and more). In this case, Monte-Carlo methods are evidently much faster.

So, the main case when it is necessary to apply parallelization is when n is large and, therefore, Monte-Carlo methods are to be used.

A mesh of size $m \times m$ contains m^2 processors. For Monte-Carlo methods,

we require either 26 or 51. The values m^2 that are close to these are $25 = 5 \times 5$ and $49 = 7 \times 7$. In both cases, the mesh has an odd size. In view of this, we will illustrate our algorithms in the following text on the example of an odd-size mesh, with $m = 2p + 1$ for some p.

For meshes of odd size, there is a processor in the center, so we will take that processor as a CP.

Coordinate system. To describe a processor in a 2-D mesh, we must describe its two coordinates X and Y. It is natural to choose the central processor as the origin of the coordinate system, i.e. as a point with coordinates $(0, 0)$. If we take the distance between the neighboring processors as 1, then coordinates of each processor are integers running from $-p$ to p. So, each processor is described by its coordinates (X, Y), where $X = -p, -p+1, \ldots, 0, \ldots, p-1, p$, and $Y = -p, -p+1, \ldots, 0, \ldots, p-1, p$.

A processor (X, Y) is directly linked to its 4 neighbors: $(X + 1, Y)$, $(X - 1, Y)$, $(X, Y + 1)$, and $(X, Y - 1)$ (whenever they exist) [1].

What each processor can do. Each processor has a CPU, and four I/O ports for transmission and reception of data. So, a processor can either be processing data, receiving data from one of its neighbors, or sending data to one of its neighbors. Each of these three possibilities demand the use of the processor's CPU and, thus, cannot be done simultaneously. For the same reason, a processor can read only one "message" (i.e., data received from only one neighbor) at a time, and send only one message at a time. However, it is perfectly reasonable to assume that a processor can transmit the same message (i.e., the same chunk of data) to all four neighbors in the same unit of time.

Time necessary for one communication step. Communication consists of elementary communication steps, i.e., of sending and receiving "messages" (chunks of data). Each sending and receiving takes approximately the same amount of time. We will denote this amount of time by t_c (c for *c*ommunication).

If one processor sends a message (= chunk of data) to its neighbor, then it takes the time t_c to send this data, and the time t_c to receive it. As a result, the entire process takes the time $2t_c$.

Remark. As a result, we arrive at the following description of what we call a *realistic* parallel computer.

2.2 Short description of a realistic computer

The realistic computer that we will consider in this paper is a 2-D mesh (grid) consisting of $(2p+1) \times (2p+1)$ identical processors. The user communicated with the central processor (CP) of the mesh: i.e., he inputs the initial (input) data into the CP and then gets the final computation results from the CP.

Each processor has 4 neighbors. At every moment of time, each processor can do one (and only one) of the following three things:

- do some computations;
- send some chunk of data to one, two, three, or all four of its neighbors;
- receive data from one of the neighbors.

The time that is required to send one block of data is assumed to be equal to the time that is required to receive this data. This time will be denoted by t_c.

2.3 The routing that is nearly optimal

A brief description of the error estimation algorithm. In order to explain how error estimation may be done in parallel, let us briefly recall the algorithm.

We assume that each processor is already equipped with a program that computes $y = f(x_1, \ldots, x_n)$ from $x_1, \ldots, x_n$. Before any computation takes place, the user sends the measured values $\tilde{x}_1, \ldots, \tilde{x}_n$ and their accuracy characteristics $\Delta_1, \ldots, \Delta_n$ (or $\sigma_1, \ldots, \sigma_n$) to a central processor CP. Then, CP communicates this input data to all other processors. As soon as each processor receives its data and has transmitted the data to its neighbors, it can start computing the corresponding value $y^{(k)}, k = 1, \ldots, N$.

Comment. The definition of $y^{(k)}$ was given in in Section 1. This definition uses random numbers. Generating a random number is a reasonably fast procedure (much faster than sending them). Therefore, we assume that each node generates its own random numbers in its computation.

After the computations are over, all processors must send their values to CP, at which time CP produces the final estimate Δ (or σ).

Let us now explain the routing (i.e., who sends data to whom, and in what order). We must describe routing for distributing the input data to all the processors and for collecting the data from them.

The lower limit on the total running time. Whatever routing algorithm we use, we need to send the input data $\tilde{x}_1, \ldots, \tilde{x}_n, \Delta_1, \ldots, \Delta_n$ (that is initially in CP) to all the processors, in particular, to a corner processor (p, p). Each processor must apply f to some data that it generates, and then all the results $y^{(k)}$ of applying f must be returned back to a CP for the final processing.

The corner processor (p, p) cannot start its computations before the input data reaches it. To reach from CP to (p, p) on a 2-D mesh, we must make $2p$ passes, and each pass takes $2t_c$. So, $(p.p)$ can start computations only after the time $2p \times 2t_c = 4pt_c$. If by T, we denote the time that is necessary to apply an algorithm f, then this corner processor will be done by the time $2pt_c + T$. To get the result of this computation back to CP, we again need at least $2p$ communication steps, so we need at least the time $4pt_c$. Therefore, we cannot get all the results into CP earlier than the moment $T + 4pt_c + 4pt_c = T + 8pt_c$. Processing these results $y^{(k)}$ also takes some time; we will denote this time by t_{comp}. For the above-described Monte-Carlo methods, $t_{comp} \ll T$. So, the final result (i.e., Δ or σ) will be ready at the moment $T + t_{comp} + 8pt_c$.

In other words, in addition to the computation time $T + t_{comp}$, we need to spend at least $8pt_c$ on communications.

This lower estimate is not attainable because we did not take into consideration *contention effects*. Namely, we can (as we will see) send the input data to all the processors, so that this initial data will reach the corner processor exactly at the time $4pt_c$. However, when we want to send the values $y^{(k)}$ back to the CP, we cannot do that without spending more time. Indeed, e.g., for a 3×3 mesh, first a CP gets the input data, then its four neighbors $(0, \pm 1)$ and $(\pm 1, 0)$. These four neighbors will be simultaneously ready with their values $y^{(k)}$, but we cannot simultaneously send them to a CP, because a CP can receive only one chunk of data at a time.

The main objective of this paper is *to produce a routing algorithm for which communication time will be as close to the above-given lower bound as possible.* Let's describe the corresponding routing.

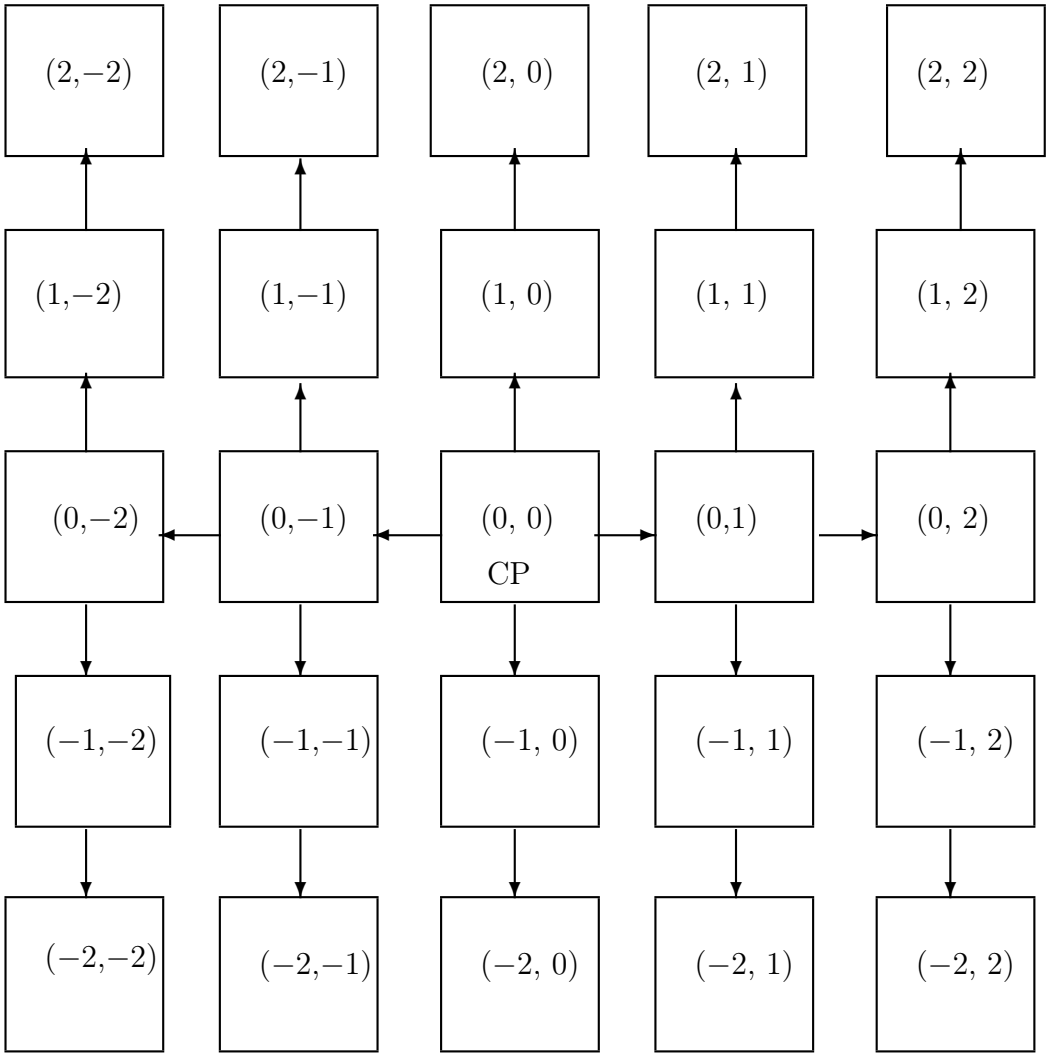

Figure 1: Routing for distribution of the initial data

Routing for distribution of the initial data (see Fig. 1 for $p = 2$).

Let us start from the moment when CP has received the input (initial) data (i.e., the values $\tilde{x}_i$ and their accuracy characteristics). We will denote this moment of time by 0. After receiving this input data, CP sends it to all four neighbors (this takes time t_c) and starts computing its value $y^{(k)}$.

Further communications are done as follows:

Central row, right-hand side (i.e., processors $(X, 0)$ with $X > 0$). Such a processor, upon receiving the input data (i.e, the values $\tilde{x}_i$ and their accuracy characteristics), sends it to 3 neighbors: up (to $(X, 1)$), down (to $(X, -1)$),

and (if $X < p$) to the right (i.e., to $(X + 1, 0)$). This CP now has all the data needed, so it can start its computation.

Central row, left-hand side (i.e., processors $(X, 0)$ with $X < 0$). Such a processor, upon receiving the input data, sends it to 3 neighbors: up (to $(X, 1)$), down (to $(X, -1)$), and (if $X > -p$) further to the left (i.e., to $(X - 1, 0)$). After that, it starts computations.

Processor (X, Y) *above the central row* $(Y > 0)$, upon receiving the input data, sends it (if $Y < p$) further up (to $(X, Y+1)$), and starts computations.

Processor (X, Y) *below the central row* $(Y < 0)$, upon receiving the input data, sends it (if $Y > -p$) further down (to $(X, Y - 1)$), and starts computations.

When do the processors start their computations? From the description of the initial routing, we can determine when each processor starts its computations. Let us first consider processors in a central row, i.e., processors with coordinates $(X, 0)$, $X \neq 0$.

Processor $(1, 0)$ receives the input data after 1 communication cycle (sending and receiving), i.e., after the time $2t_c$. It spends t_c time to send this input data to its neighbors $(2, 0)$, $(1, 1)$, and $(1, -1)$. Hence, at time $3t_c$ it starts computing.

For a processor $(X, 0), X \neq 0$, it takes $|X|$ communication cycles for the input data to reach it; then one more communication step to send the input data to its neighbors. Hence, it starts computing at the moment $(2|X|+1)t_c$.

Let us now consider an arbitrary processor (X, Y), $Y \neq 0$. According to our routing algorithm, the input data that is coming to this processor first goes to the processor $(X, 0)$ in the central row, and then follows the $X-$th column. Therefore, we need $|X|$ communication cycles for the input data to reach the processor $(X, 0)$ and $|Y|$ additional communication cycles for the input data to reach the desired processor. This takes $2(|X| + |Y|)$ communication steps. If a processor is not on the edge of the mesh $(|Y| \neq p)$, then we need one additional communication step during which this processor sends the input data to its neighbors; only after this step, the processor can start computing.

So, we arrive at the following conclusion: a processor (X, Y), $|Y| \neq p$, starts computing at the moment $(2|X| + 2|Y| + 1)t_c$; a processor (X, Y), $|Y| = p$, starts computing at the moment $(2|X| + 2|Y|)t_c = (2|X| + 2p)t_c$.

When do the processors finish their computations? Since we assume that all the processors are similar, and since they are all doing practically the same task (apply the algorithm f to some values), we may conclude that each of them takes the same amount of time to perform its computations. Let us denote this amount of time by T.

Therefore, when $|Y| < p$, a processor (X, Y) finishes computations at time $T + (2|X| + 2|Y| + 1)t_c$, and when $|Y| = p$, at a time $T + (2|X| + 2p)t_c$.

Main idea: overlapping communications and computations. As we have mentioned, the nodes closer to CP finish their computations earlier. We are going to take advantage of that while collecting data. Namely, to achieve ultra performance, we will start collecting the results $y^{(k)}$ from the computers that have already finished their computations while the farther nodes are still computing.

Remark. We are greatly thankful to the anonymous referee, who formulated this idea in very clear terms, and thus helped us to improve both the algorithm and its exposition.

Routing for collecting the data from the processors: main ideas. We have already mentioned that we cannot just send all the results $y^{(k)}$ right into CP. For example, processors $(1, 0)$, $(0, 1)$, $(-1, 0)$, and $(0, -1)$, finish their computations at the same time. If they all try to send their results to CP, we will get contention. So, we need some routing.

General idea: first collect data from each column into a central row, then collect data from this central row into a CP. To avoid contention, we will first collect data for each column (X, Y), $-p \le Y \le p$, into a central processor $(X, 0)$ of that column. At that time, processors from the central row will have all the data. Next we will collect the data from them into CP.

Data collecting in each column and in the central row will be a two-stage process. Processors $(X, 1)$ and $(X, -1)$ will finish their computations at the same time. To avoid this contention, for each column, we will first collect data into two processors: $(X, 0)$ and $(X, -1)$, and then merge these collected pieces of data together.

Similarly, when we collect data from the central row, we first collect the data from the right-hand side processors into $(0, 0)$, and the data from the left-hand side processors into $(-1, 0)$, and then merge these pieces of data.

Resulting routing algorithm.

Upper half ($Y > 0$). When a processor (X, Y) finishes its computations, it sends its result to its lower neighbor $(X, Y - 1)$. If it receives a result from its upper neighbor, it sends it unchanged to its lower neighbor.

Lower half ($Y < -1$). When a processor (X, Y) finishes its computations, it sends its result to its upper neighbor $(X, Y + 1)$. If it receives a result from its lower neighbor, it sends it unchanged to its upper neighbor.

Row -1 ($Y = -1$). When a processor $(X, -1)$ finished its computations, it waits for data to come. When data comes from below, it merges this data with what it already has. After receiving $p - 1$ messages, it waits for $4t_c$ (to avoid contention, see below), and sends the resulting collected data to its upper neighbor $(X, 0)$.

Central row, right-hand side ($X > 0, Y = 0$). When a processor $(X, 0)$ finishes its computations, it waits for data to come. When data comes, it merges this data with what it already has. After receiving $p + 1$ chunks of data (p from $(X, 1)$ and 1 from $(X, -1)$), it sends the data it collected to its left neighbor $(X - 1, 0)$.

Central row, left-hand side ($X < -1, Y = 0$). When a processor $(X, 0)$ finishes its computations, it waits for data to come. When data comes, it merges this data with what it already has. After receiving $p + 1$ chunks of data (p from $(X, 1)$ and 1 from $(X, -1)$), it sends the data it collected to its right neighbor $(X + 1, 0)$.

Processor $(-1, 0)$. When a processor $(-1, 0)$ finishes its computations, it waits for data to come. When data comes, it merges this data with what it already has. After receiving $2p+1$ messages (p from $(1, 1)$, 1 from $(1, -1)$, and $p - 1$ from $(-2, 0)$), it waits for $4t_c$, and then sends the collected data to CP (0,0).

CP. When processor $(0, 0)$ finished its computations, it waits for the data to come. When data comes, it merges this data with what it already has.

After receiving $2p+2$ messages (p from $(0,1)$, 1 from $(0,-1)$, p from $(1,0)$, and 1 from $(-1,0)$), it uses the collected data (i.e., values $y^{(k)}$, $1 \le k \le N$) to estimate the desired value Δ or σ, and sends this value to the user.

What time does it take to collect all the data: Part 1. Collecting data from a column. Let us first trace the process of collecting the values $y^{(k)}$ from the nodes in a column X. This trace is illustrated on Fig. 2.

In this figure, states of a column in different moments of time are depicted from bottom to top, so that the first row depicts the first moment of time, etc. In the lowest row, numbers (-3), ..., 3 denote y-coordinates of the nodes. A node that has already finished computing its value $y^{(k)}$ is marked by crossing it over. An arrow from one node to another means that the data has just been sent. On top of each node, we describe what data are already collected in this node. E.g., "0,1,2,3" means that this node at this moment of time contains the results $y^{(k)}$ of computations of nodes with y-coordinates 1, 2, and 3. At the last moment of time, when all these results are collected in the central processor of this column, we mark this central processor by ***.

$t = T + 2|X|t_c + t_c$. The first processor to finish computations in this column will be the central processor $(X,0)$ of this column: it will be done in the moment $T + 2|X|t_c + t_c$.

$t = T + 2|X|t_c + 3t_c$. At this moment, processors $(X,1)$ and $(X,-1)$ are done. Processor $(X,1)$ sends its result to $(X,0)$. Sending and receiving takes $2t_c$.

Remark. While this sending is going on, other nodes (that are farther away from CP) are still computing their respective values $y^{(k)}$. So, this routing algorithm really enables us to overlap communications and computations.

After a $2t_c$ time interval, we have the following:

$t = T + 2|X|t_c + 5t_c$. At this moment, $(X,0)$ has already collected 2 results: its own and that of $(X,1)$. At that very moment of time, two more processors are done: $(X,2)$ and $(X,-2)$. They start sending their data correspondingly to $(X,1)$ and $(X,-1)$ (meanwhile other nodes are still computing).

...

$t = T + 2|X|t_c + 2pt_c$. The last processors to finish their computations are (X,p) and $(X,-p)$. They will be done by the moment $T + 2|X|t_c + 2pt_c$. Processor (X,p) sends its data to $(X,0)$. This sending requires p intermediate steps, each of which takes time $2t_c$. So, the total time that

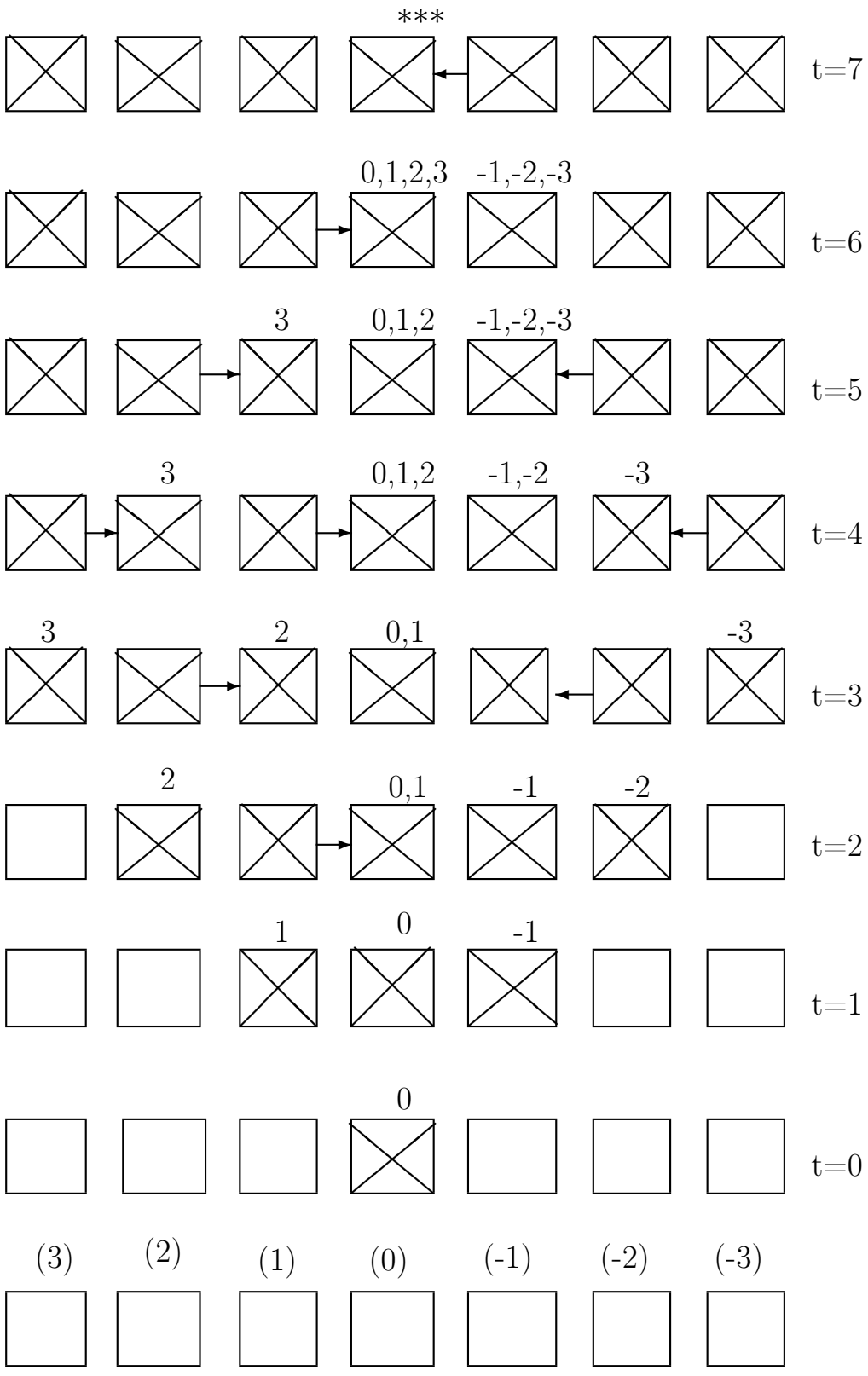

Figure 2: Collecting data ($y^{(k)}$) from a column

is required for the value $y^{(k)}$ computed by a processor (X, p) to reach the central processor $(X, 0)$ of this column is $2pt_c$. Therefore, this value reaches $(X, 0)$ by the time $T + 2|X|t_c + 4pt_c$.

...

$t = T + 2|X|t_c + 4pt_c$. By this time, all the values $y^{(k)}$ computed by the processors (X, Y) with $Y \geq 0$, are already collected and stored in in $(X, 0)$, and all the values $y^{(k)}$ computed by the processors (X, Y) with $Y \leq -1$, are already collected and stored in $(X, -1)$. So, the only thing that remains to be done in order to collect all the data from the entire column the central processor $(X, 0)$ of this column is to send the data collected in $(X, -1)$ to $(X, 0)$. This sending takes time $2t_c$, so after that time interval, we are done with this column.

$t = T + (2|X| + 4p + 2)t_c$. All the data from the column is in $(X, 0)$.

What time does it take to collect all the data: Part 2. Collecting the data from the central row.

This procedure is somewhat similar to the process of collecting the data from one column, so we can still use Fig. 2 as an illustration, with the understanding that rows in Fig. 2 now depict the states of the central row in different moments of time.

$t = T + (4p + 2)t_c$. According to Part 1, the first column to finish the collection of all its computational results is column 0; it finishes the collection exactly at this moment of time.

$t = T+(4p+4)t_c$. At this moment, columns 1 and -1 are done (i.e., have also finished collecting their results into correspondingly $(1, 0)$ and $(-1, 0)$). So, processor $(1, 0)$ sends all the values that it has collected so far to $(0, 0)$. Sending and receiving takes $2t_c$. So, after another $2t_c$ time interval, we have the following:

$t = T + (4p + 6)t_c$. At this moment, $(0, 0)$ already has all the computation results from 2 columns: its own (0) and column 1. At that very moment of time, two more columns have finished collecting their computation results into one processor: all the results of column 2 are collected in the processor $(2, 0)$, and all the results computed by processors from column -2 are collected in the processor $(-2, 0)$. The central processors of these columns start sending the data that they collected correspondingly to $(1, 0)$ and $(-1, 0)$, etc.

...

$t = T + (6p + 2)t_c$. The last columns to finish collecting the results of their computations are p and $-p$. They will be done by the moment $T + (6p + 2)t_c$. In other words, at this moment of time, the computation results $y^{(k)}$ obtained by all the processors (p, Y) in p-th column are collected in the processor $(p, 0)$, and the computation results from the processors $(-p, -p)$, ..., $(-p, 0)$, ..., $(-p, p)$, are collected in the processor $(-p, 0)$. Processor $(p, 0)$ sends the values that it has collected to CP $(0, 0)$. This sending requires p intermediate steps, each of which takes time $2t_c$. So, the total time that is required for the collected data to reach CP is $2pt_c$. Therefore, this collected data reaches $(0, 0)$ by the time $T + (8p + 2)t_c$.

...

$t = T + (8p + 2)t_c$. By this time, all the values from the processors (X, Y) $X \geq 0$, are already collected and stored in CP, and all the values from the processors (X, Y) with $X \leq -1$, are already stored in $(-1, 0)$. So, the only thing that remains to be done in order to collect all the data from all the processors into CP is to send the data collected in $(-1, 0)$ to CP. This sending takes time $2t_c$, so after that time interval, we are done.

$t = T + (8p + 4)t_c$. All the data from the all the processors is in the CP.

Result: what is the total communication time? All the results of all the processors are in CP at the time $T + (8p + 4)t_c$. By t_{comp}, we denoted a time that is necessary to process all these data ($t_{comp} \ll T$); so, the total time when we can give the result to the user is $T + (8p + 4)t_c + t_{comp}$.

In other words, in addition to main computation time T, and time t_{comp} that is necessary for processing the results $y^{(k)}$ of simulations, we spend $(8p + 4)t_c$ on communications.

This result shows that our routing algorithm is close to being optimal. We have already determined that the user cannot get the final results before the time $T + t_{comp} + 8pt_c$.

Our result was a little larger because we had to delay some communications in order to avoid contention. However, this delay is reasonably small: $4t_c$ in addition to the ideal $8pt_c$. Let us illustrate the relative values of our estimate and theoretical lower bound $8pt_c$ for two examples: $p = 5$ and $p = 7$.

$p = 5$**:** our algorithm gives $44t_c$, while the lower bound is $40t_c$. So, our communication algorithm gives only a 10% increase over the lower bound for communication time.

$p = 7$**:** our algorithm gives $60t_c$, while the lower bound is $56t_c$. So, our communication algorithm gives only a $\approx 7\%$ increase over the lower bound for communication time.

In terms of N. The total number N of simulated values $y^{(k)}$ is $m^2 = (2p+1)^2$. In terms of N, $2p+1 = \sqrt{N}$, $p = (\sqrt{N}-1)/2$, and so *the total communication time is* $4\sqrt{N}t_c$.

3 Three-dimensional mesh

For a 3-dimensional mesh, with $(2p+1) \times (2p+1) \times (2p+1)$ processors (X, Y, Z), each of which is directly connected with 6 neighbors $(X \pm 1, Y, Z)$, $(X, Y \pm 1, Z)$, and $(X, Y, Z \pm 1)$, we may apply similar routing techniques. Namely, when we spread the initial data around, each processor $(0, 0, Z)$ sends this initial data to all 6 of its neighbors, and all the other processors send the initial data only to other neighbors in the same XY-plane (i.e., only to the neighbors with the same value of Z coordinate). When we collect the data, we apply the above-described 2-D routing to collect the data from each XY-plane into its central processor $(0, 0, Z)$, and then (just like we collected data from the central row into CP), collect the chunks of data that has already been collected into the processors $(0, 0, -p)$, ..., $(0, 0, p)$, into CP.

One can show that for a 3-D mesh, communication takes time $(12p+6)t_c$, which is also pretty close to the lower bound $(12pt_c)$. In terms of $N = (2p+1)^3$, this time is equal to $6\sqrt[3]{N}t_c$.

4 Experimental verification on an emulated machine, and hopes for real-life implementation

In this paper, we presented a parallel algorithm for a realistic parallel machine, and (theoretically) analyzed its performance. To verify our analysis of performance, we implemented this algorithm on an emulated 2-D mesh (emulated using a transputer board for a PC [4]). The entire algorithm, including routing and computing of f, was written in Occam [3]. This test was repeated for several different functions f. To check whether we correctly computed the timing of all the steps, we made each processor in addition to computing f, to compute a (global) time, so that before sending or receiving any message, a node waited until the global time coincided with the one that we estimated above. This way, we may have sometimes overestimated the running time, because in reality, the times that different nodes require may be slightly different. But with this overestimated numbers, we could check that the description of the routing was consistent, and our counts of the numbers of communication steps were correct: the total running time was equal to $T + t_{comp} + 8pt_c$.

These emulations prove that our theoretical estimates are correct, but since they are not done on a real parallel machine (only on its emulation on a PC), they are not yet real parallelization results. Unfortunately, we do not have a real 2-D or 3-D mesh at out disposal. But our theoretical results, and this additional emulated verification makes us hope that this algorithm will be used for real-life parallel computations. We will be glad to collaborate with anyone interested in that.

Conclusion

Many algorithms that estimate error of indirect measurements (e.g., methods based on numerical differentiation, or Monte-Carlo methods) are easily parallelizable. The resulting parallel algorithms do not require extensive communications, we may achieve a great speed-up.

On the majority of actual parallel computers, however, any communication is a very time-consuming procedure. If we wish to estimate errors

of indirect measurements using these computers, we must develop routing algorithms that eliminate message contention, and thus, reduce the total communication time. In this paper, we propose such algorithms for a 2-D and 3-D meshes. We show that for these routing algorithms, communication times are close to being optimal.

References

[1] Akl, S. and Lyons, K. A. *Parallel computational geometry.* Prentice-Hall, Englewood Cliffs, N. J., 1993.

[2] Hwang, K. and Briggs, F. A. *Computer architecture and parallel processing.* McGraw-Hill, N. Y., 1984.

[3] INMOS, ltd. *OCCAM 2 reference manual.* Prentice-Hall, Englewood Cliffs, N. J., 1988.

[4] INMOS, ltd. *Transputer development system.* Prentice-Hall, Englewood Cliffs, N. J., 1988.

[5] Kreinovich, V., Bernat, A., Villa, E., and Mariscal, Y. *Parallel computers estimate errors caused by imprecise data.* Interval Computations 1 (2) (1991), pp. 31–46.

[6] Moore, R. E. *Methods and applications of interval analysis.* SIAM, Philadelphia, 1979.

[7] Villa, E. *BaRe: a multiprocessing system using broadcast and replication.* Master Thesis, The University of Texas at El Paso, 1991.

E. Villa
Department of Mathematics
El Paso Community College
El Paso, TX 79998
USA

A. Bernat, V. Kreinovich
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968
USA