INFORMATION PROCESSING 89, G.X. Ritter (ed.)
Elsevier Science Publishers B.V. (North-Holland)

CONTRIBUTIONS OF THEORETICAL COMPUTER SCIENCE, APPLIED COMPUTER SCIENCE AND NUMERICAL MATHEMATICS TO THE DESIGN OF PARALLEL COMPUTERS

S.M. MÜLLER and W.J. PAUL

Lehrstuhl für Rechnerarchitektur und Parallele Rechner
Universität des Saarlandes, D-6600 Saarbrücken, F.R.G.

1. Why is it so hard to teach relations between theory and practice ? We feel that there are three roots for this.

The first is that research in theoretical and in applied computer science is done in very different ways. Theoretical computer science is a (well funded) branch of pure mathematics. Models are simple. Factors of 2 for cost or performance are of little importance. Complexity of proofs and the introduction of new concepts are by themselves of value. Applied computer science is engineering. Paper designs alone are of little value. One builds prototype systems in order to find out if things really work and if nothing important has been overlooked in the original paper design. Complexity is bad. Ingenuity tends to add confusion and should only be applied where absolutely necessary to achieve the design goals.

Second improving one's skills in serious engineering is of little help for doing serious theorem proving and vice versa. Hence anybody trying to earn his living as a mathematician/engineer will have in reserach a hard time against the specialists in both fields. Because professors are hired mostly on the basis of their research there are hardly any mathematician/engineers teaching at universities.

Third theory is of course of use to build better systems (otherwise there would be no problem; we would ignore theory) but the process is not obvious. Theory besides producing theorems produces language to talk about computer systems. This language is often used by designers when they have to document their systems and this process works well. But theory every now and then produces theorems which can be turned into useful systems or which help understand real systems better. Turning theorems into systems or explaining things about real systems with theory is hardly ever taught. It requires sound knowledge of theory and systems, but university professors tend to be specialists.

2. Theoretical and practical considerations in the design of parallel computers. We sketch some mixed (i.e. theoretical/practical) arguments which could be presented in a class about architecture of parallel computers.

Suppose we want to build a parallel machine. We study PRAMs and simulations thereof and find that they are presently not parctical. So we settle for processors with private memory which are interconected via a network to be determined.

Theoretical machines of this kind work in pairs of rounds. In the first round all processors do one step of local computation. In the second round each sends and/or receives one item of data over the network. Because in real systems there will be startup time for communication and delay across the network we change this. We maintain the simple concept that in each round all processors either compute locally or they communicate. However computation rounds involve many steps and they last until the last processor is done. In communication rounds processors send and/or receive arrays rather than single data items. System software to support this is trivial. Studying lots of applications from numerical mathematics and physics shows that this very simple scheme is adequate unless one gets into some very fancy dynamic load balancing.

The dream communication network is a crossbar. It has the disadvantage of costing $O(n^2)$ switching elements for n processors. Because the number of gates is an established measure of cost of circuits in switching theory we believe that we should save switching elements. We settle for some nice network from theory with $O(n \log n)$ switching elements and a hypercube like interconnection structure. This network allows us to simulate the crossbar in most situations we need.

During the design the hardware of the network cables and connectors enter the szene. They are (fortunately) absent in switching theory and ugly and unreliable in real life. We don't like them and pack the network on a small number of boards. One of the boards ends up really big, it has lots of layers and few active components. Theory of VLSI circuits helps us understand. Laying out a hypercube on a VLSI-chip requires area $\Omega(n^2)$. The arguments are applicable to printed circuit board area.

We find that cost of circuit boards is dominated by area unles they contain some very fancy chips. We might have been better of to stick to a crossbar right away. Theory has brought us confusion and insight. Manufacturers are not better off. Some sell parallel machines with crossbar like backplanes and they reassure us correctly that the crossbar can simulate a hypercube.

3. An argument on the granularity of parallel machines. The theory of parallel algorithms is strongly inspired by the theory of efficient algorithms for sequential machines. This theory tries to find fast algorithms for certain model problems like sorting, graph connectivity, multiplication of natural numbers, travelling salesman etc. There is consensus that the techniques to solve these model problems efficiently will enable a programmer in general to write efficient programs for sequential machines (and for the single processors of parallel machines). But it is by no means clear that efficient solutions for these same model problems on parallel machines provide the techniques to write in general efficient programs for parallel machines.

We have studied a different set of model problems. Because presently the typical applications of real parallel machines are in physics simulations and numerical mathematics we studied how to parallelize the algorithms which are presented in university classes on basic numerical mathematics and on numerical solution of partial differential equations.

Our model of computation are p processors with private memory interconected by a crossbar. We counted floating point operations and memory access with cost 1 and all other operations with cost 0. Transmitting an array of n floating point numbers over the crossbar costs $n + S$, where S is the startup time for communication. One of the bad surprises of building real parallel computers is that typically $S = 100$ due to system software.

It turned out that the vast majority of algorithmes could be paralellized on many processors. We made the following observations.

The algorithms operate typically on large $n \times n$-arrays (matrices or grids), where n is around 1000 in case the array is dense and n is around 10^6 in case the array is sparse. In any case the number N of elements to be processed in the array is around 10^6. The single processors perform the sequential algorithm on appropriately chosen parts of these arrays and they communicate. They work in single program multiple data mode but very often not in SIMD mode. Communication serves typically only three purposes:

i) exchanging borders of the processor's part of the array with his neighbors ,
ii) collecting global information along the edges of a tree (e.g. global sums) and
iii) broadcast of global information.

Load is typically balanced if each processor has N / p elements of the array. Between communication each data item contributes a constant cost say α. The constant α is typically between 1 and 50.

The border of a square with N/p elements has length $4\sqrt{N}/\sqrt{p}$. Other geometries make the relative length of the border larger or make the load unbalanced. Thus exchanging data with neighbors costs typically $4\,(S + \beta\,\sqrt{N}/\sqrt{p}\,)$. The constant β is usually 1, 2 or 4 depending on the width of the border strip which is exchanged and if processors can send and receive simultaneously.

Collecting global data is best done along the edges of a balanced binary tree. One will get $2 \log_2 p$ rounds of communication. If in each round g data are sent along the edges which are used cost is $(2 \log_2 p)\,(S + g)$. Usually $g \leq 10$.

Broadcasting usually does not consume significant time. Let t_p be the time the parallel program needs for one round of local computaion ond communication. Let t_s be the time the serial program needs for the corresponding part of the computation and let eff be the effectivity of the parallel program. Let $\gamma = 0$ if no data are transmitted along borders and let $\gamma = 1$ otherwise. Let $\delta = 0$ in case no global information is collected and let $\delta = 1$ otherwise. Then we get

$$t_s = \alpha N \ , \qquad t_p = \alpha N/p + 4\,\gamma\,(S + \beta\,\sqrt{N}/\sqrt{p}\,) + \delta\,(2 \log_2 p)\,(S + g) \ ,$$

and

$$\mathrm{eff} = t_s / (p\,t_p) = \alpha N / (\,\alpha N + 4\gamma p S + 4\beta\gamma\sqrt{N}\,\sqrt{p} + (2\delta\, p \log_2 p)\,(S + g)\,)$$

We define the granularity G of the processors as the number of items of the array a single processor can handle in his local memory. Then $p \geq N/G$. Substituting $p = N/G$ in the equations above gives

$$\mathrm{eff} = \alpha N / (\,N(\alpha + 4\gamma S/G + 4\beta\gamma/\sqrt{G} + 2\delta\,(S + g) \log_2(p)/G$$

$$\geq 1 / (1 + 4S/G + 16/\sqrt{G} + 2(S + 10) \log_2(p)/G)$$

Setting $S = 100$ we get

$$\mathrm{eff} \geq 1 / (1 + 400/G + 16/\sqrt{G} + 20 \log_2(p)/G\,)$$

Let $p = 1024$. If $G = 1\,000$ we get $\mathrm{eff} \geq 0.24$ and if $G = 10\,000$ we get $\mathrm{eff} \geq 0.7$. In order to be cost effective with a fine grain machine ($G = 1000$) one would have to produce a fine grain processor at roughly 1/30'th the price of a medium grain processor ($G = 10\,000$). Moreover the network for the fine grain machine would be more expensive.