The Impact of Massively Parallel Computers on Image Processing

Azriel Rosenfeld  
University of Maryland

The use of massively parallel computers for image analysis was first suggested in a paper entitled "A Computer Oriented Approach Toward Spatial Problems," which appeared in the proceedings of the IRE, if I remember correctly after 30+ years. It was suggested that if you took an image and loaded it into a bit and then connected an array of processors—one pixel per processor, perhaps—you could do an awful lot in a small number of computational steps using pixel parallelism. I want to start off from that baseline and say something about what kind of things we want to do with images once we get them loaded into a massively parallel system, and how hard some of those things are going to be. We know how to massively parallelize some operations; but, we don't necessarily know where the bottlenecks are. I have my own prejudices, and I'll comment on them as I go along.

Let's look at real-time vision. You point a camera at a scene. You get a video image, grab a frame, digitize it, and now you have this massive data set, but the frames keep coming at you. You're dealing with perhaps many megabytes per second, depending on frame rate and size. One of the serious limitations in our business is that you often have to do things in real time. These limitations have no meaning, however, if you get an image every day from somewhere out in space, and you are willing to take 24 hours to make some decision about it. For most systems, we don't have that kind of luxury. In most real-world applications, for example, if you are trying to create eyes for a robot that actually has to move around and manipulate things in real time, then implementing what the system has to do in real time at low cost is a challenge.

Let's go into the factory and look at the industrial machine vision systems that are being sold. They don't work with massive parallelism, because massive parallelism is not something you can put on a chip for a few thousand dollars so that you can install them economically. So, they work with the sort of simple operations that you can perform sequentially more or less at frame rates, which tremendously limits what they can do. The things that computer vision researchers were inventing, exploring, and developing 20–30 years ago, you can now do in real time on a single-processor system.

Massive parallelism offers the tantalizing promise of being able to do less trivial things in real time once the cost of massively parallel systems comes down. The axiom is that computer power is getting cheaper, and this trend will continue. We are far from being up against a stone wall.

There are experts on hardware here who will tell you how soon you'll be able to buy a 1,000-x-1,000 mesh-connected system for $1,000. It'll be awhile, no doubt. But, unless the economic pressures for doing it go away, it would surprise me if, by the turn of the century (plus or minus a few years), we weren't within shooting distance of that target, if not already there. This implies that although we keep coming up with ever more diabolical ideas about how to torture
What types of data do we need to process when we try to do computer vision? What stages do we go through from the time we get the image into our retina (so to speak) until the time we are able to do things? What types of operations do we need to perform on these types of data, and in what way can we speed up those operations using parallelism?

It's fashionable to say nowadays that computer vision has two major goals: constructing a map of the environment, and recognizing the objects.

The catch for goal 1 is that the environment is three-dimensional (3-D), while the camera image is two-dimensional (2-D). To map the environment, you need surface topography. If I pointed a camera at this room, there's a lot of depth here, lots of objects occluding one another, and they are very complex objects.

With regard to goal 2, recognizing the objects, lets look at the people in this room. Can I recognize and count them? Can I tell the men from the women? Can I distinguish which ones are wearing eyeglasses or beards? Such recognition tasks are beyond the state of the art. You might come up with a technique that recognizes 80—90% of them, but, that doesn't mean you're doing it right—and when you blow it, you really blow it. Maybe the slide projector down the aisle will show up as a guy wearing a funny hat because your technique doesn't know about slide projectors and projector stands.

Recognition is a very hard and open-ended problem. Naturally, there are simple domains in which we can do topographic recovery; but, very little of the topographic recovery stuff has successfully been done robustly on real data. Much of it gets demonstrated on synthetic examples. Vision problems are not easy. We're starting with an image and we want to end up with certain products, or outputs, one of which is a depth map. Many people are using range sensors nowadays instead of TV cameras, although they're very slow, because they give you a depth map. Recognition is still a mess, though, because objects are 3-D. You only see one side of them. They hide one another. Objects may not even be precisely defined. (Give me the precise definition of the human head or, for that matter, a precise definition of a beard. Heads and beards come in many varieties.)

I want to stress the data types involved in vision. If you are trying to get from the pixel array to the depth map, which is also a pixel array (the pixel means something else), the data types involved are primarily pixel arrays, and you might actually get away with pixel parallelism, with what we might call "retinoptic" processing, involving processes that look at local patches of the data, chew them up, and spit them out again in array form in a way that is now more meaningful: it's now a depth map instead of an image.

In this situation, you are basically processing arrays of data; the basic data type is the array. The kinds of operations you're performing are primarily local operations where you look at little pieces of the array and infer the logical topography by massaging them. If this was all we wanted to do, it would substantiate the contention that the big bottleneck in doing vision quickly is the massive local processing of all those pixels in parallel. Mesh-connected machines like the Massively Parallel Processor (MPP) might, in fact, be the basic solution. And, even though these machines, in their pres-
ent states, still have some limitations, they’re developing and improving all the time.

When you come to recognition tasks, however, it’s fairly clear that you need other types of data representations. What’s a human face? It’s got eyes, eyebrows, nostrils, lips, and so on. In order for it to be a face, however, the parts have to be in the right places. So, now we are talking not just about images as pixel arrays, but about image parts. We’re talking about eyes being almond-shaped, with pointy ends. We’re talking about noses being aquiline—that even sounds 3-D. We’re talking about mouths being pursed or smiling. We’re talking about image parts—about geometric properties of those parts, and relations among the parts.

So, when we want to do recognition, we’re not just talking about pixel arrays; We’re talking about other kinds of data and data types—other kinds of information about these data types, and other kinds of processing of these data types.

A computer vision system may be confronted with a variety of data types. It certainly starts with array data and, at the very beginning, a particular numeric array—that’s the pixel array. It may go from there to do all kinds of derived arrays. Some of them may no longer be numeric; some of them may be symbolic and look like overlays. Even in the domain of numeric arrays, which may not even be scalar valued. These arrays might represent surface orientation. They might represent textural information, which is painfully gathered on a local basis in the neighborhood of each pixel. In short, there are many array-like representations.

Above and beyond that, we must eventually start extracting geometric entities in two and three dimensions from the pixel array. Now, we are confronted with how to represent geometric entities—patches of the image, patches of surface, or pieces of solid. These are entities in two dimensions, two-and-a-half dimensions (surface patches), and pieces of solid, not all of which you can see. A vision system must deal with the representation of that kind of information and its processing.

Going up to a still higher level of abstraction, how do you represent this data collection about pieces of the image and their properties and relations? The old standby is that you create some kind of labelled graph in which you represent the image pieces as nodes. The graph then tells you how they are related. That’s how you get from the array to some sort of abstract structure. The reason for doing this is object recognition. The description of the object is in terms of parts and their relations. So, somebody has to get that kind of information out of the image and check it against the models—the descriptions of what a generic thing is going to look like. Modeling is hard. Description is hard. Making them meet halfway so that you can check one against the other is hard.

There’s an even more abstract data type that we might call “knowledge.” I won’t even try to speculate how easy or hard it is to do vision in an AI-ish [artificial intelligence-ish] context in which you can reason about what you are doing. The processes of extracting parts from an image are not very AI-ish. The processes of setting up the data structures are not very AI-ish. People are attempting to make use of AI-ish control structures in doing some of the higher level massaging of the more abstract data types. Parenthetically, I would contend that if the AI-ish approach is going to do the vision community any good, they ought to start using it, even down at the pixel level. But,
since I'm not prepared to prove that speculation, let's pass it by.

The real thing I want to call your attention to is to not assume that the bottleneck is only at the pixel level. Yes, the data you begin with in your vision system are pixel arrays of various sorts, whether the original one or all sorts of derived ones. Yes, those arrays involve fairly massive amounts of data. For example, 1,000-x-1,000 pixel images give rise to arrays of 1 million pixels, and if we have 1 or a few bytes per pixel, we're talking 1 or a few million bytes of data. Yes, that's a lot of data. But massive parallelism is approaching the million level, even if it hasn't quite gotten there yet. (Maybe you'll hear product announcements at this meeting.) We already know how to break the bottleneck at the pixel level for certain types of operations through massive parallelism, because, as massive as the parallelism may be, we're on our way toward it. And, once we achieve that level, when the parallelism of the machine is equal to that of the problem, then we can process every pixel in the image simultaneously. If all we're trying to do is some kind of local processing, maybe repeated local processing (local means not very large neighborhood sizes), of the pixel arrays, massive parallelism of the conventional kind—mesh-connected machines—would allow us to do it very fast. Thus, if the bottleneck occurred at the pixel level, conventional massive parallelism would break it.

The question is, what about the possible bottlenecks at the more abstract levels? Here, an optimist might say that, at these levels, we're working with fragments of the image; and how many of them are there? Perhaps just a few hundred. But how many bytes does it take to tell us everything we want to know about one of these fragments? Not a vast number. Then, why do I insist that there may be problems up ahead?

Because we may run into a combinatorial explosion. True, I said that from your million-pixel image, all you need do is extract, say, 1,000 image fragments. What gets you into trouble is that you pull out these 1,000 fragments in 100 different ways. Anybody who thinks you can run one canonical segmentation technique on an image and get the definitive thousand atomic image fragments is wrong. You need to extract the fragments in many different ways. So, in fact, they represent possibly overlapping inconsistent interpretations of pieces of the image. Then, you need to put those fragments together in combinations. True, you're almost certainly not considering arbitrary combinations of the thousand-image fragments (2-to-the-thousandth-power combinations); you're probably looking only at certain connected combinations, and although I'm not prepared to count them, it's certainly not a fully combinatorial problem. But, there are still many combinations, and that's where the true bottleneck may lie.

What kinds of operations do we want to perform on these various data types? There is a taxonomy; it's a kind of textbook of basic image processing and analysis techniques organized by type of operation.

If I'm given a pixel array, I might want to work on it one pixel at a time. I might want to do a stretch of the gray scale or a thresholding. I might want to do a huge variety of local operations. These are a generalization of point operations, where we're not just operating on single pixels, but one pixel and a few other related pixels everywhere in the image. It's obvious how to do that kind of
thing in a massively parallel way, but some kinds of things get a little less obvious.

Suppose I want to do statistics on the image, perhaps to analyze its texture. How do you get global statistics on a 1,000 x 1,000 image? Not by local operations. Somehow, you've got to get all the information together in one place, so you can count noses, so to speak. (How many occurrences of some particular local property are there in the entire image?) The mesh doesn't support that too well. A 1,000 x 1,000 mesh looks like it's giving you a millionfold speedup factor in processing, but that's for local or point operations. It's giving you only a 1,000-fold speedup in statistical computation, because you still have a communication problem.

Other kinds of image transformations provide other problems. There are geometric image transformations that perform arbitrary warping of our image to correct distortions. There are other kinds of transformations in which the output is still an image, but it no longer has even a geometrically distorted point-by-point correspondence with the input image. Finally, there is the large class of segmentation operations that perform the segment extraction of the image parts. The input is an array, but the output no longer is.

Suppose we have managed to pull out of our million-pixel image 1,000 image fragments, or something on that order, and we have somehow represented them (without giving a lecture on representations), so we now have descriptions that are sufficient to reconstruct each of those fragments. In other words, we have a collection of geometric entities. What sorts of things do we want to do with them?

It starts right out as combinatorial in that we want to assemble them in various ways.

I may need to take unions of collections of them. I may need to intersect some of them. I may need to derive other subsets from them. I don't think there's a general agreement on the taxonomy on what you may want to do with image parts. What are the geometric computations you need to perform? And how can they be efficiently performed? What are the good ways of representing the geometric entities?

Yes, you only have 1,000 entities; but, you may need to deal with a very large number of combinations of the entities. And every time you form a new combination, you may have to recompute everything, especially since the images are coming along at 30 per second. Whatever it is you do, you may have to do it again, especially if things changed rapidly. On the very next frame, you may have to extract and/or combine fragments, a different combination every time, and then compute derived structures, geometric properties, and decide on geometric relations of all sorts on the resulting mass of data.

We have accumulated a lot of ideas over the past decades as to the types of things we need to do. We have reasonably efficient algorithms for doing them. We can now look at this body of tasks and ask how can we speed them up? Is massive parallelism useful when you are trying to handle the combinatorics of search with the goal of combining image parts so that you can get to the next stage of description?

Similar remarks are true at the next level of abstraction, the graph level, where we have thrown away the geometric details. A geometric entity is now represented by a graph node at the location of the entity. But graphs are combinatorial objects, too. Even at the graph level, you get into the combinatorics of considering collections of nodes, and the complexity gets at least polyno-
mial. I'm not a graph theorist. I'm not asking what the taxonomy of computations is that you want to do on labelled graphs. I'm only asking what a taxonomy of labeled graph computations is that a vision person might want to do.

When you look at vision benchmarks nowadays, you find that the creators of the DARPA vision benchmarks deliberately stuck their necks out and said “What about computational geometry and graph algorithms?” The next DARPA architectures workshop will be held next week, in Avon, CT, where people will report on a unified benchmark involving operations at all the different levels of abstraction.

To summarize: What's the vision problem? What are we trying to do? Topographic recovery? Object recognition? What sorts of data do we need to process? The pixel array? Derived arrays of all kinds? Geometric objects, represented in various ways? Still more abstractly, labeled graph structures? Beyond that, I don’t even want to suggest anything.

What kind of operations do we want to perform on these data types? We have a long list for the array types, a shorter list for the geometric types, and a still shorter list for the graph types. I don't claim that the list is really shorter, but only that I’ve been too lazy to think harder and come up with a convincing, definitive taxonomy of what we may want to do. Now comes the question, “What about the speedup of these operations using various forms of parallelism?”

The pipeline idea yields operation parallelism. No sooner do you finish doing an operation on the first little piece of your image, then, treading right on its heels, comes another processor that starts the next operation on the partial output of the first operation. By doing this, you’re overlaying operations; if you can do one operation at frame rates, then you can do K operations, not in K times the frame rate, not K times slower; by overlapping, you can do K operations practically at frame rates.

The MPP, and the other mesh-connected parallel systems, provide another approach, which allows an operation to be performed in parallel at every pixel. The tree-structured machines represent something that you might call orthogonal to the mesh. It’s a different interconnection structure, but tremendously powerful for certain purposes. If you want to do statistics operations—for example, if you want to do histogramming, a tree is great.

A pyramid is basically the cross-product of the mesh and the tree. It has the advantages of both. I’m not going to give you a pep talk on pyramids, but, it should be clear from the program of this meeting, and almost any meeting these days, that pyramids are currently undergoing a wave of popularity, which is nice.

Finally, there's a wave of commercial machines using hypercube architectures. Hypercubes—if they’re sufficiently massively parallel—are very advantageous. In terms of communication flexibility, they can simulate pyramids very handily.

These architectural ideas have been around for some decades. The mesh has been around in a conceptual way since 1958, so it's having a 30th anniversary. Eventually, a succession was built. People have been talking about hypercubes for 10–20 years. People have been talking about pyramids for 10–15 years. Using all these kinds of parallelism—a lot is known about that. But the vision problem is still a challenge.
Suppose I was showing slides instead of these dull black-and-white, alphanumeric transparencies. Suppose I hit the slide changer button and up on the screen appeared a slide of an octopus. How long would it take you to recognize it? A fraction of a second. Suppose I hit the button, and the next slide is the Eiffel Tower. Again, in 1/10th of a second, you think “Eiffel Tower.” They are familiar objects. A typewriter is familiar. My Doberman Pinscher (which I don’t have) is familiar. And so on. You never saw that particular dog before, and maybe you don’t know breeds that well; but, in 1/10th of a second, you recognize it as a dog. It’s a familiar object. You didn’t expect it, but you recognize it in a fraction of a second. You have a long-term memory in your head. As a child, you learned at an incredible rate. It’s been estimated that a child learns to recognize 5,000 objects by the age of 10. (This is based on counting the entries in a picture dictionary.) You can name more than 5,000 objects reliably. You have all this information stored in your head. And you can instantly access it in a fraction of a second. How much processing could have gone on in your head from the time the light hit your retina to the time the word “octopus” came to the surface?

Neurons are slow; they take milliseconds. There is controversy as to exactly how neurons do their computations (computation is the best metaphor we have these days), but, whatever it is they do, and however they encode it, and however it is represented computationally—somehow, there is something going on in your head that goes from the light hitting your eye with an octopus pattern to the word “octopus” coming up in your short-term auditory memory, and coming out as a word.

When we do that in a few tenths of a second, uncoached, unprompted, and unexpected,