What P2P Computing Is and Is Not:
Why Elepar Has No Peer in P2P Computing

Elepar, Draft

July 5, 2001
www.elepar.com

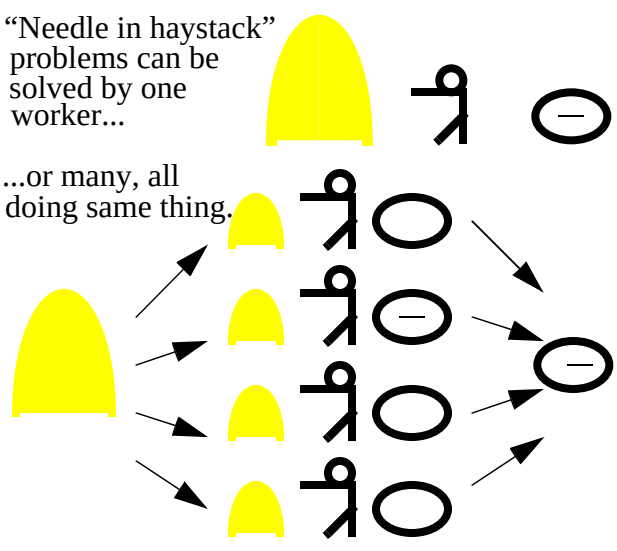
There has been much press over the last year or so about “peer-to-peer” computing. Fueled first by Napster, and then by new start-ups and some large computing companies, the casual observer might imagine that one can take any program and execute it with supercomputer speed by simply acquiring a P2P product from any one of these companies and then enlisting the use of PCs around the office and/or internet. This is certainly not true. This document is an attempt to add some light to the heat by characterizing the state of the art in very simple terms, and contrasting Elepar’s approach with that of the rest of the field.

How most P2P computing systems work

“P2P” (peer-to-peer) is a term that became popular to describe music and information transfer approaches, like Napster. The power of these approaches is in the ability for user’s PCs or workstations, “peers”, to contact one another directly, rather than to funnel all interaction through a central server. Some P2P file transfer systems, like Napster, still rely on a central server for finding out which peers have the data one needs, or the peers needing the data one has, but the ultimate value of P2P comes after that, when the peers contact one another directly for the real work.

In addition to just sharing information, one can imagine peers sharing in the solution of a computational problem by interacting directly with one another. In fact, this idea has been around for decades, with names like “parallel”, “distributed”, or “concurrent” computing. Interestingly, now in the age of P2P, few companies are taking advantage of this technology. In fact, in most current “P2P computing”, the peers never communicate with one another, but only with a server, so the true power of P2P is never being harnessed.

The majority of current P2P approaches only work with certain kinds of programs, such as those that start with a very large set of input data (or a large set of input parameters) and perform the same sorts of computations to all of it, a piece at a time. These programs are similar to the old problem of finding a needle in a haystack. If you were to do it, you would need to perform the same action (looking for something that looks like a needle) no matter which part of the haystack you looked in, and once you were done looking in a particular place, there would be no reason to look at it again, or to remember anything about what you saw for later, other than whether it held a needle or not.



So, you could recruit all of the friends you wanted to help you look, as long as you could explain the task to them, and make sure that all parts of the haystack were covered. (Even if some parts were covered multiple times, it would be OK.) There would be no need to tailor the task to each individual worker.

This is exactly the kind of problem, and the *only* kind of problem, that most existing P2P computing approaches handle. In fact, they are largely borrowing their software directly from the "Search for Extra Terrestrial Intelligence" (SETI@Home) project which is almost literally looking for a needle in a haystack, only the haystack is the sky, and the needle is some signal that looks like the result of intelligence. For these products, a server divides up the haystack, the peers each process their portion, and the server collects and processes the answers. Peers don't contact peers.

The world is not a haystack

Just as most problems in daily life are not like looking for a needle in a haystack, so most computing problems are not either. Maybe the program needs to remember some data it looked at a long time ago to combine with new data. Maybe it really is looking through a haystack, but for something which requires a much wider view to find, so one "peer" can't see it alone. Maybe it needs to simulate a large activity where things happen over relatively long periods and/or over wide areas. Maybe it just needs to react to things which are happening even as it runs. Whatever the reason, most programs are not "needle in haystack" problems, so these P2P approaches are no help.

There are other solutions being marketed for P2P that are a little more flexible. One is based on "replicated objects", which is almost the opposite extreme of client server. It also revolves around the idea that the peers will be performing similar tasks to one another, but instead of just having each communicate an answer back to a server when they are done, virtually all peers inform virtually all other peers of much of what they each do. Designed primarily for gaming systems where all players need to see the same game state, it may work well in some cases, but as you might guess, when there are a large number of peers, and each really only needs a small amount of information from a small number of other peers to do its job, it will probably use significantly more overhead and bandwidth than is really required.

Another approach is "distributed objects", in which the programmer explicitly breaks down the program into individual components (objects) while it is being written. In fact, the programmer must decide a lot of things when it is being written, like which parts will probably end up on different peers and which parts will end up on the same peers, as well as what kinds of security each object should have, etc. The problem is, much of this will depend on circumstances that cannot be known when the application is written. And, even if it could be known, getting optimal performance and correct results while dealing with these other issues is extremely difficult. A more primitive form of this same basic approach is "peer-pipes", which just help to feed the output of one program as input to another (on a different peer), and are very limited in their application.

What's the solution? First ask, "What's the problem?"

To understand the extent of the issues involved in P2P computing, first imagine that you run a huge organization, with offices in several cities, and many people at each office, maybe each having approximately equal skillsets. You also have a huge task that needs to get done as soon as possible. How do you use your personnel to get the *right* answer in a relatively short time?

In traditional parallel and distributed computing, the standard approach has been akin to first expressing the task as though one person was going to do it ("sequentially"), then getting a roster

of all the people (or "peers") in the organization capable of doing the work, and painstakingly deciding what each will do, and when. This is obviously a nightmare of micro-management, just figuring out how to break the problem down into the right number of pieces, that will truly get the right answer and keep people moderately busy, all while they shuffle intermediate results (think memos or email) between themselves. If something goes wrong, tracing down the problem can be nearly impossible. And as hard as that all is, now consider that after you get it right, work begins, and then some people get sick, or other priorities arise, or one of your offices has a crisis (e.g. fire). You not only need to figure out who else can do their work, but where the former workers were when they left off, what kinds of internal notes they were keeping that are needed to make progress, and whether those notes are even still accessible.

As difficult as this is, given the assumptions above, it is still at least feasible in some cases—e.g. tasks which consist of steps, each of which is somewhat similar to a "needle in a haystack". In that case, you can tell everyone to look for this needle, then make some decisions based on that, then ask everyone to look for the next needle, etc. Even these cases are traditionally carefully planned so that everybody finishes in about the same amount of time to minimize idle waiting. This is possible, in part, because of the assumptions above—i.e. that the number and geography of the workers are known, that they are all at management's disposal, and they all work at about the same rate.

Of course, all of those simplifying assumptions are nowhere to be found with P2P. With P2P, nobody knows how many peers will be available, how proficient they will be, how they will be situated to one another, whether they all speak the same language, or whether some of them will be trying to accomplish other work at the same time. Because the peers may come and go, just finding what peers are available, and getting basic information about them, may be a task in itself that must be performed even while the main task continues. (In fact, in the very worst case, some peers could actually be your competitor, either spying on the computation or intentionally providing false intermediate results, but this is a case where an ounce of prevention—i.e. restricting the peer group—is worth a pound of cure—i.e. figuring out who's the "mole".) One can begin to understand why current P2P solutions are only of the most basic variety.

Grinding it up

Think of the compute power of each peer as a bucket, and the program as a big chunk of computation that needs to be split among them. If you know exactly how many buckets you have, and how big each is, you can try to break the computation into exactly the right number and size of chunks to fit one chunk per bucket. But since we've already said that, in P2P, the number and size of buckets can change even as the program runs, a more rational approach is to virtually grind up the computation into smaller pieces, or "grains". That way, we can just pour the grains into the buckets as they come along, putting in as many as will fill that particular bucket.

However, a computation is not just an inert substance that can be haphazardly ground up: More like a chemical reaction or an ecosystem, it consists of pieces that depend on one another. Going back to the office analogy, grinding up the computation means that instead of giving each worker one big personal task description, we basically have a "job jar" in each office, of much smaller tasks that we can spread around to whoever is available, but each task will still require partial results from other tasks. We would most like a worker to pull a job from the jar only if that job already has all of the necessary intermediate results ("inputs") clipped to it, so that the task can be performed from beginning to end without the need to wait for more intermediate results. (This analogy also illustrates that "grinding too fine" can have its own problems, with workers spending

too much time just fiddling with the job jar.)

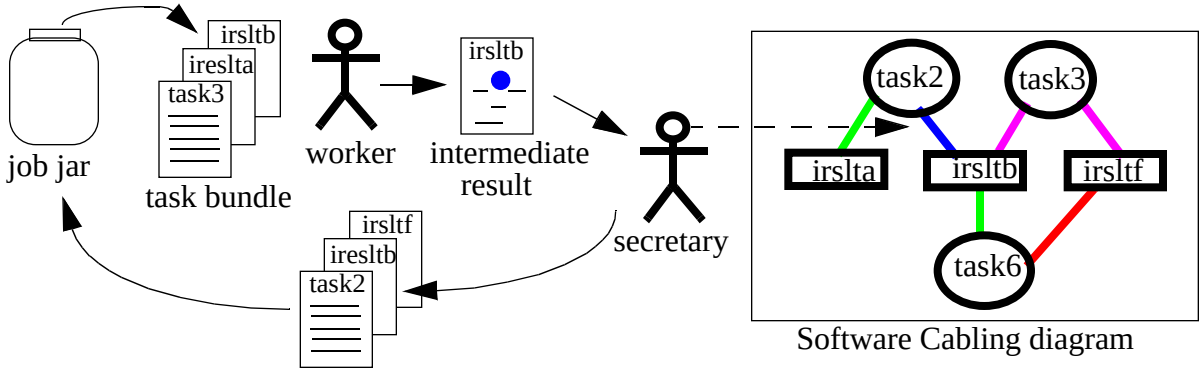
"Grinding up" a sequential computation into small pieces is very tough work. This may be counter-intuitive, because virtually all programs are created from smaller pieces in the first place (with names like "functions", "subroutines", "objects", "modules", etc.), which are composed by the software engineer, in a hierarchical and/or peer-like fashion, to create the ultimate software product. The problem is that, for the most part, these compositional approaches were developed decades ago, and are a one-way street: Once the pieces have been composed together, they are no longer distinct (or at least, not separable) in the running program--they must all remain together, on the same computer ("peer"). Also, the composed pieces take turns running, so even if it was possible to split them onto different peers, only one of those peers would be active at a time.

All of this prompts questions. Instead of "grinding up" software, is it possible to adapt traditional software composition approaches so that the original pieces from which it is created remain distinct, and the "one at a time" rule is relaxed, resulting in software that is automatically in the form of these small tasks for job jars? Even if so, how do the results from one worker's task get to the correct job jar and clipped to the correct bundle to be drawn out later? And with all of those little tasks and intermediate results, how does one keep track of everything?

Elepar's Software Cabling (or SC)

As the result of extensive research over the last 15 years, Elepar's founder has devised Software Cabling (or "SC"), a new, graphical way of composing pieces of software together in a way that many pieces can run at a time on different peers, or they can all end up on one. The individual pieces can be written in virtually the same way, and with the same languages and compilers, that are already used. Regardless of the language(s) used for the pieces, SC has many powerful features of its own that help to engineer very large and efficient software project. (In computerese, these go by names such as hierarchical and object-oriented development styles, software templates, data parallelism, and function-based program analysis.)

SC employs some simple rules and software, analogous to an efficient secretarial staff in each office building, to make P2P work. In the analogy, when a worker has produced an intermediate result (even if the entire task is not complete), he/she puts a colored dot on its sheet of paper, as directed by the task description. The secretarial staff then looks at the color of the dot, and *only* the color of the dot, and consults an SC diagram to see what to do with it, if anything. Briefly, that SC diagram has a circle for each task, a rectangle for each intermediate result, and colored lines connecting each task to the intermediate results that it will use or produce. The secretary finds the rectangle corresponding to this intermediate result, finds the connecting line matching the color of the dot, sees the task at the other end, and does whatever is necessary to make sure that the inter-



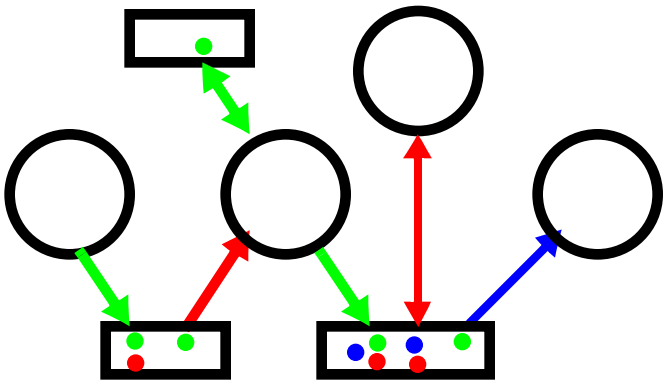
mediate result gets attached to the bundle corresponding to that task, wherever it may be (removing the colored dot in the process). The secretary can also translate the documents in the process, if the workers speak different languages. And, if the secretary keeps a safe copy of intermediate results until the new task completes, then even if one worker fails to complete that task for any reason, the task bundle can be recreated to give to another worker (known as "fault tolerance").

Note that, just as in the needle in haystack cases, the workers here can be completely oblivious to what other workers (peers) are doing, or even if there *are* other peers, but unlike the needle in haystack cases, the overall effect is that intermediate results are flying back and forth between peers. As long as there are completed bundles in the job jar—i.e. there is work to do—workers keep busy. This is in stark contrast to cases where the programmer manually splits the overall task among the peers. In that case, not only must a peer often sit and wait while awaiting an intermediate result from another peer (or look for other work after painstakingly recording exactly where it left off), it must also be aware of where intermediate results are expected from and/or going to. SC is much more flexible if the user decides to just run it on one peer, or peers disappear.

SC's simple and efficient process keeps everyone busy doing what they do best. For example, a task description may tell the worker to take an input sheet with lots of information, to just change a few numbers, and to put a colored dot on it. The secretary will then take it and bundle it onto the next task. In other computing approaches, someone—often the worker—would be required to copy all of the data to a new sheet, changing just the few numbers, and then personally figure out where the new sheet would go next, addressing the envelope and all. Also, note that if the next task is in the local job jar, there is very little overhead: Just remove the colored dot, bundle, and done. In other approaches, the worker is required to effectively place each piece of paper into an envelope and label it, etc., *just in case* it needed to be shipped out to another office.

The big picture

From the explanation above, there's no obvious advantage to having the secretary look at a diagram (with circles, lines, and rectangles) instead of, say, a textual form containing the same information. It becomes clearer when you consider that the people designing and/or maintaining the program need to understand how it all works together, and that includes how intermediate results flow among all of these tasks. In fact, SC diagrams contain other annotations that make this even more apparent. For example, the lines (between the task circles and the intermediate result rectangles) have arrowheads that tell whether the worker will just be looking at the intermediate result, just producing it, or updating the information that's already there as a new intermediate result. Also, colored dots within the rectangles show the possible colored dots that might end up on each intermediate result as the result of executing that task. Together, these annotations not only help a programmer to get a bird's eye view of the overall workings of the program, they also



When you learn to read the SC diagram, you can see not only how intermediate results can flow from one task to another, but the possible order that tasks can be performed in based on the dot colors they produce

help the secretarial staff to plan and optimize motion of the intermediate results.

Remember that each worker produces results based only on the task description and the intermediate results which were clipped to the task bundle when it started. To figure out what a task is going to do, or how a peer is going to act, you don't need to be concerned with anything else that happened (or is happening) in any other task. In computerese, tasks that act like this are known as "side-effect free" or "functional", and they make the program much easier to debug and to analyze (i.e. to help determine if it will really do what it is supposed to). Even though there can be a lot going on at once, both the programmer and the peer only need to focus on one thing at a time.

Understanding a program with dozens (or maybe thousands!) of tasks would certainly be a problem if all of the tasks were represented in one diagram. SC diagrams themselves can be built in pieces,thereby allowing even the largest and most complicated programs to be built and represented visually. (SC calls these diagram pieces "boards", and the tasks themselves "chips", in an analogy to the way a computer itself is built.) In addition to aiding in comprehension, piecing up SC diagrams like this also makes it easy to use techniques from common and popular programming approaches, like "object-oriented programming", when building very large SC diagrams.

Elepar is not the only entity recognizing the utility of designing large programs graphically. One popular current approach is called Unified Modeling Language, or UML. While SC was not designed to compete with UML, likewise UML was not designed to address many P2P problems described here. Also, UML actually uses as many as seven different kinds of diagrams for each program, depending upon what sort of information one wants to see, while one SC diagram contains all of the most important information: Nothing is hidden, except for the specific computer code within each task, and even that can always be summarized as a function. And although tools exist to help create a program from a set of UML diagrams, the diagrams can go out of date as the program evolves. For SC, the diagram is always up-to-date, because the diagram *is* the program.

But, even given all of these nice features, there are still times when one might not want to deal with color or graphics. Maybe the program must be represented in a medium that is optimized for text and/or black and white, or one wishes to use more traditional text-based tools, like a text editor, when performing certain operations to the program. Maybe the programmer is simply color blind, or wishes to store away the program in a space-efficient form that can be read by a human without special tools. For these reasons, even though the SC program always looks like a colored graphical diagram when the programmer is working with it, it is always stored onto the disk in a human-readable textual form (called ESCORT, for Elepar's SC Open Representation as Text).

Elepar's other solutions

Elepar understands that some software developers may be wary of using a new software development method like SC. They may want to break up their programs between peers themselves, the old fashioned way, but they still see the value in that efficient "secretarial staff" that optimizes data motion. For these people, Elepar has designed Cooperative Data Sharing, or CDS. As its name implies, CDS lets different tasks share data efficiently, consistently, and flexibly, regardless of whether those tasks are on the same computer or different computers, performing translation and encryption if necessary.

Then, there is the whole process of finding, scheduling, and paying the workers to be involved. For this, Elepar has developed an approach called "PICA", or "People, Instruments, Computers, and Archives". PICA is basically a societal marketplace model for providing, finding, and paying

for computational goods and services without resorting to micropayments. Elepar does not currently have products that specifically facilitate PICA, but the design serves as a backdrop against which other Elepar products are being created. More info on all is available at www.elepar.com.

Conclusion

All programs are valued first and foremost for their functionality. If a program has intrinsic distributed functionality, then P2P technology may be mandatory, but in other cases, possessing a P2P execution mode, while useful, will not in itself be a significant reason for buying (or for creating) a program. And programs that run only in a P2P mode may also be deemed limited.

It follows that it will not be feasible for software developers to separately create, sell, and maintain both a "standard version" and a "P2P version" of every program. A comprehensive development approach (supported by quality tools) must be used, that is at least as efficient, productive, flexible, and professional as existing ones, for developing one program which runs in both P2P and traditional settings. Whenever possible, it must allow the use of the languages and techniques which have been developed over the decades to facilitate the creation of quality software in any number of different fields.

It is also clear that because of the very dynamic nature of P2P networks, more of the work in deciding what computation will happen where must be done automatically by the computer, rather than by the programmer, because the information needed to make these decisions effectively is not even available until the program is actually running. Elepar believes that it is time to bite the bullet, and accept that the world is changing, being careful not to go overboard by throwing out perfectly usable technologies. By adapting the entire software development process to realistically account for the parallelism, fault tolerance, security, dynamic loads and heterogeneous architectures found in P2P computing, rather than trying to address them as an afterthought, the software engineering process can actually be simplified and improved. In doing so, the real result is broadened from creating a new P2P methodology to one for writing efficient, easy to understand, very portable programs that can execute in a variety of settings, including traditional sequential, high-performance parallel, *and* P2P.