A Theory of Parallel Computation
The π-calculus

Background

- DFAs, NFAs, pushdown automata, Turing machines... All are mathematical entities that model computation. These abstract systems have concrete, practical applications in computer science (CS).

  For example, deterministic finite automata (DFAs) are associated with regular expressions, which computer programs that involve pattern matching frequently rely on. Also, knowing theoretical results such as the inability of any computation to determine whether or not another computation will stop (Halting Problem) can keeps programmers from attempting to write impossible computer programs.

- Automata represent one approach to mathematically modeling computation. There are others.

  For example, the mathematical logician Alonzo Church created a formalism of computation based on functions in the 1930s, called the *λ-calculus*. The key notion in this approach is an operator (i.e., function) called *λ* that is capable of generating other functions.

  One of the earliest high-level programming languages, LISP (for LISt Processing language, 1959), is a practical computer implementation of the λ-calculus. LISP was designed originally for research in *artificial intelligence (AI)*, a field in CS that perpetually seeks to extend the capabilities of computers to carry out tasks that humans can do. Scheme and Clojure are some contemporary programming languages descended from the original LISP, and and other widely used "functional" programming languages such as ML and Haskell are based on the λ-calculus. Programmers use these languages to develop useful applications, and researchers use them to explore new frontiers in computing.

- From a theoretical viewpoint, the λ-calculus embodies all essential features of functional computation. This holds because the relationship between "inputs" (domain values in Mathematics, arguments/parameters in programming) and "outputs" (range values in Math, return values in programming) from functions expresses everything in a purely functional system of computations (no "state changes"), and λ-calculus is the mathematical theory of functions considered entirely according to their "inputs" and "outputs."

  In fact, it can be proven that any other foundation for *functional* computation, such as Turing machines (which can express any type of computation), will have exactly the same expressive power for functional computation as the λ-calculus [Pierce 95].

- However, all of the computational models we've mentioned so far (Turing machines, λ-calculus, etc.) are for *sequential computations* only. This means that we assume only a single computational entity. Until a few years ago, it was reasonable to assume that only one computational processor would be available for most computations, because most computers had only one computational circuit for carrying out instructions.

- Nowadays, retailers sell only *multi-core* computers (i.e., computers having multiple circuits for carrying out instructions) on the commodity market, and hardware manufacturers such as Intel and AMD no longer produce chips with only one computational processor. This results from computer engineering having reached certain limitations on performance for individual processors (related to electrical power consumption, access to computer memory, and parallel speedup capabilities with a single processor).
- Consequently, the only way to continue improving the performance of computers going forward is to use *parallel computing*, in which multiple computer actions are carried out physically at the same time. Parallel computing (or *parallelism*) can be accomplished by writing programs that use multiple computational cores at the same time, and/or by running multiple cooperating programs on multiple computers.
- Some computations are easy to parallelize. For example, a computation may involve applying exactly the same program steps to multiple independent input data sets, in which case we can perform parallel processing by executing that series of program steps on multiple processors (i.e., multiple cores and/or computers), and submitting different data sets to different processors. We call this strategy *data parallelism*. Some authors refer to such computations as being *embarassingly parallel*.
- Other types of computations may be parallelizable without being data parallelizable. For example, matrix multiplication requires combining the rows and columns of rectangular arrays of numbers in ways that require accessing each number multiple times, in different groupings. Parallelization strategies for matrix multiplication exist, such as multiplying submatrices formed by subdividing the original matrices, then combining those results appropriately. However, those strategies are more complex than simple data parallelism.
- Many computations require parallelizing according to the computational steps instead of (or in addition to) parallelizing according to the data. When a computation has multiple processors carrying out different sequences of computational steps in order to accomplish its work, we say that computation has *task parallelism*.

  For example, imagine a computation that extracts certain elements from a body of text (e.g., proper names), then sorts those elements, and finally removing duplications. With multiple processors, one might program one processor to extract those elements, another to perform the sorting operation, and a third to remove the duplications. In effect, we have an assembly line of processes, also called a *pipeline* by computer scientists.

- Computer scientists have found other computations exceedingly difficult to parallelize effectively. Notably, nobody knows how to parallelize finite state machines (FSMs) well, as a general class of computations. [View from Berkeley 06, p.16]
- We can easily imagine how to construct a mathematical model of computation for simple data parallelism from a model of computation for the sequential case of that same computation, by replicating the sequential model. This approach seems promising as long as we can assume that those multiple parallel computations do not need to interact with each other in any way.
- However, more complicated forms of parallelism that involve multiple processes interacting in various ways, such as the task parallelism example of pipelining, requires a mathematical model of parallel computation capable of expressing those interactions between processes.

  The π-calculus, introduced in the next section, is an example of such a model of parallel computation.

The π-calculus, informally

- A *calculus* is a method or computation based on symbolic manipulation.
  - In *differential calculus*, symbolic manipulations involve an operator $\frac{d}{dx}$ that satisfies rules such as the following:
    - $\frac{d}{dx}(f+g) = (\frac{d}{dx}f) + (\frac{d}{dx}g)$
    - $\frac{d}{dx}(f \cdot g) = (\frac{d}{dx}f) \cdot g + f \cdot (\frac{d}{dx}g)$
  - In *integral calculus*, symbolic manipulations involve an operator $\int \ldots dx$ that satisfies rules such as the following:
    - $\int f + g\,dx = \int f\,dx + \int g\,dx$
    - $\int f \cdot (\frac{d}{dx}g)\,dx = f \cdot g - \int (\frac{d}{dx}f) \cdot g\,dx$
  - In the λ-calculus, symbolic manipulations involve an operator λ that has its manipulation rules, involving operations such as substitution of variables and applying functions to particular "input" values (function calls).
- The operators and manipulation rules for a calculus may have useful concrete applications. For example, the differential calculus rules are satisfied by certain continuous mathematical functions, where the operator $\frac{d}{dx}$ represents the rate of change of those functions.

  We typically think of $\frac{d}{dx}$as operating on those functions, although the differential calculus rules are actually abstract and might be applied to other entities than functions.

- The π-calculus has six operators. We think of them as operating on *sequential processes*, i.e., running computer programs, although they are abstract and can be used without any particular concrete application.
  - The *concurrency operator* $P|Q$ (pronounced "*P* par *Q*") may be thought of as two processes *P* and *Q* executing in parallel (e.g., simultaneously on separate cores or on different computers).
  - The *communication operators* may be thought of as sending and receiving messages from one process to another, across a communication *channel* that is used only by those two processes (i.e., a *dedicated* communication channel in the language of CS).
    - The *output prefixing* operator $\overline{c}\langle x\rangle.P$ (pronounced "output *x* along *c* (then proceed with *P*)") may be thought of as send a message *x* across a channel *c*, then proceeding to carry out process *P*. Here, the channel *c* may be thought of as starting from this process to another.

      Channels such as *c* may be set up between any two processes, but those two processes are then uniquely determined for *c*, and may not be changed later. Channels provide for a single communication in one direction only, specified when the channel is created.

    The "dot" that appears in this notation indicates the boundary between one step and a next step in a process.

    - The *input prefixing* operator $c(y).P$ (pronounced "Input *y* along *c*") may be thought of as waiting to receive a value from the channel *c*, and once a value is received, storing that value in *y* and proceeding to carry out process *P*.

  - The *replication operator* $!P$ ("bang *P*") may be thought of as creating a new process that is a duplicate of *P*.

    This sort of an operation is quite realistic in parallel computing. For example, a *web server* is a program that receives requests for particular web pages and responds by sending those web pages. Web servers must be capable of handling multiple responses at the same time, because some web pages may take a significant amount of time to prepare and deliver, and it would be undesirable for one user to be delayed by another user's request. Therefore, a web server system may start up a new duplicate process for handling each request it receives. (Students who have studied operating systems will also see an analogy between the system call fork() and this replication operator.)

    In the π-calculus, arbitrarily many duplicate processes are created by a single application of the replication operator.

  - The *name allocation operator* $(\nu c).P$ ("new *c* in *P*") may be thought of as allocating a new constant communication channel *c* within the process *P*. The symbol ν is the Greek letter *nu*, pronounced like "new".
  - The *alternative operator* $P + Q$ ("*P* plus *Q*") represents a process capable of taking part in exactly one alternative for communication. That process cannot make the choice among its alternatives; that selection among alternatives cannot be determined until it occurs, and once determined, any remaining alternatives have lost their chance and will never occur. (These restrictions on the alternative operator are not strictly necessary for π-calculus to work, but they simplify the theory.)
- Besides these operations, there is one constant process 0 that does nothing. For example, we might write $\overline{c}\langle x\rangle.0$ for a process that sends one message across a channel *c*, then does nothing more.
- Observe that all of the operations have to do with entire processes or with communication among processes. For example, there are no arithmetic operations such as multiplication, nor any operations related to applying (i.e., calling) functions, nor a way to store values in memory (assignment). The π-calculus is entirely concerned with communication among processes that are executing in parallel.

  However, a theory of sequential processes, such as automata or the λ-calculus, can be used in conjunction with π-calculus in order to model both the parallelism of communication and sequential algorithms that take place between communication events.

In our examples, we will use an informal notation for the sequential aspects of a process for readability and convenience, but we will use the π-calculus formalism carefully in matters of parallelism and communication between processes.

Here is an example that models parallel computation using the π-calculus operators.

A *client-server application* is a parallel system in which a program running on one computer, called the *server* program, responds to requests that may be sent by programs that may be running on other computers, called *client* programs. One example of a client-server application consists of web browsers (as clients) communicating with a web server (as server). However, there are other possibilities.
Consider a client-server application in which clients send requests to a server to apply a particular function to arguments that a client provides. In CS, this type of service is called*remote procedure call (RPC)* (where "procedure" is another term for "function"). RPC can enable clients to obtain the results of computations that those clients may be unable to compute on their own "local" hardware.

We will model RPC using a simple incrementing function.

- Here is C++ language code for the desired function.

  ```
  int incr(int x) {
    return x+1;
  }
  ```

  In case you are not a programmer: The first line indicates that the name of this function is incr, and that incr accepts one integer input (argument) named x and returns an integer value (as indicated by the int at the beginning of the line). The second line is a return statement, which specifies the output ("return value") in terms of the inputx. This incrementing function returns the value x+1.
- Here is a model for the server process:

  $$!incr(c,x).\overline{c}\langle x+1\rangle.0$$

  Here, the expression *x*+1 indicates sequential code, but the remainder of the expression uses π-calculus formalism. Observe that *incr* is a channel for communicating to the server.
  The use of the replication operator ! means that the entire remainder of the expression will be duplicated as many times as needed (in order to serve as many RPC requests as may arrive over time). We will consider the operator ! to have higher *precedence* that | and + but lower precedence than the other π-calculus operators; this means that the expression above is equivalent to

  $$!\big(incr(c,x).\overline{c}\langle x+1\rangle.0\big)$$
- Here is C++ code for part of a client process:

  ```
  y = incr(17);
   ...
  ```

  The dots represent steps to be taken after accomplishing a remote procedure call of *incr*.

  Note for non-programmers: in this C++ context, the symbol = is an *assignment operator*, not an equality relation. The effect is to compute the result of applying the function *incr* with input value 17, and to store the output (return value) into computer memory under the name y.

Note for everyone: The mathematical effect of making an assignment is substitution. In other words, the assignment of 18 to *y* means that every occurence of *y* should be replaced by 18 throughout the program steps indicated by dots above.

  - Here is a model for that client process, starting from the assignment above:

    $$(\nu a)\big(\overline{incr}\langle a, 17\rangle.0 | a(y).P\big)$$

    Here, we create a new channel *a* and *send that channel*, together with the value 17 that we want to increment, to the server, using the *incr* channel from client to server. The channel *a* is for communicating from the server back to the same client. Observe that the output along *incr* requesting the service takes place in parallel with the input along*a* for delivering the result. (Of course, the first of these will necessarily occur before the second in this particular situation.) The entire client model consists of π-calculus expressions, except for the integer 17.

    In this expression, the process *P* represents steps the client will take after the remote procedure call of *incr*. In other words, *P* represents the dots in the client code above. We want RPC to cause *y* to be replaced by 18 throughout *P*.

  - We can now express a model for the entire client-server application.

    $$!incr(c, x).\overline{c}\langle x + 1\rangle.0 \quad | \quad (\nu a)\big(\overline{incr}\langle a, 17\rangle.0 \mid a(y).P\big)$$

- *Structural congruence*, an equivalence relation on π-calculus expressions.
- *Reduction*, the "calculus rules" for π-calculus.
- We can now use the definition of structural congruence and the reduction rules to give a formal proof that our π-calculus model of an *incr* remote procedure call service produces the results we desired for it.

$$!incr(c,x).\overline{c}\langle x+1\rangle.0 \quad | \quad (\nu a)\big(\overline{incr}\langle a,17\rangle.0 \mid a(y).P\big)$$

$$\equiv \quad incr(c,x).\overline{c}\langle x{+}1\rangle.0 \quad | \quad !incr(c,x).\overline{c}\langle x{+}1\rangle.0 \quad | \quad (\nu a)\big(\overline{incr}\langle a,17\rangle.0 \mid a(y).P\big)$$

by structural congruence axiom for !
(this dispenses a copy of the server process to use)

$$\equiv \quad !incr(c,x).\overline{c}\langle x{+}1\rangle.0 \quad | \quad incr(c,x).\overline{c}\langle x{+}1\rangle.0 \quad | \quad (\nu a)\big(\overline{incr}\langle a,17\rangle.0 \mid a(y).P\big)$$

by commutative law for |

$$\longrightarrow \quad !incr(c,x).\overline{c}\langle x{+}1\rangle.0 \quad | \quad \overline{c}\langle x{+}1\rangle.0[c,x/a,17] \quad | \quad (\nu a)\big(0 \mid a(y).P\big)$$

by main reduction rule (this corresponds to sending a message)
Note: the notation [*c*,*x*/*a*,17] means to replace *c* by *a* and replace *x* by 17.

$$= \quad !incr(c,x).\overline{c}\langle x+1\rangle.0 \quad | \quad \overline{a}\langle 18\rangle.0 \quad | \quad (\nu a)\big(0 \mid a(y).P\big)$$

by definition of substitution and arithmetic

$$\equiv \quad !incr(c,x).\overline{c}\langle x+1\rangle.0 \quad | \quad \overline{a}\langle 18\rangle.0 \quad | \quad (\nu a)\big(a(y).P \mid 0\big)$$

by commutativity axiom for |

$$\equiv \quad !incr(c,x).\overline{c}\langle x+1\rangle.0 \quad | \quad \overline{a}\langle 18\rangle.0 \quad | \quad (\nu a)\big(a(y).P\big)$$

by identity axiom for |

$$\longrightarrow \quad !incr(c,x).\overline{c}\langle x+1\rangle.0 \quad | \quad 0 \quad | \quad (\nu a)\big(P[y/18]\big)$$

by main reduction rule

$$\equiv \quad !incr(c,x).\overline{c}\langle x+1\rangle.0 \quad | \quad (\nu a)\big(P[y/18]\big)$$

by associativity and identity for |

In this proof, we started with the π-calculus expression for the server *and* the π-calculus expression for the client *before* RPC, running in parallel. We ended with that same server we began with, and with a client process *P after* RPC that has every occurrence of *y* replaced by 18 -- as desired.

Exercises

1. If *a* does not appear in *P*, show that the last line above is structurally congruent to ! *incr*(*c*,*x*).*c* *x*+1 .0 | *P*[*y*/18]. Give a formal proof segment using the axioms and reduction rules. (NOTE TO WORKSHOP PARTICIPANTS: the red c should have a bar over it.)
2. Prove the following facts, using formal proofs from axiom and reduction rules, as in the verification of the RPC server above.
   a. $0|P \;\equiv\; P$
   b. $!P \;\equiv !P|P$
3. Write a π-calculus expression that models an RPC system for an echo function, whose return value (output) is the same as its argument (input).

   Hints: Modify the RPC example for incr to serve echo instead. You can use the same client expression as before, but you will need to alter the server expression. Since the problem asks for a *system* instead of only a server, your final answer should be a π-calculus expression for *both* the client and the server.

Here's a C++ programming language definition of echo, in case it's helpful.

```
int echo(int x) {
  return x;
 }
```

4. Examine the formal proof of the π-calculus model of an incr RPC service above, and indicate how to transform it to a proof of your π-calculus model of an echo RPC service in the previous problem.
   Suggestion: It might be convenient to print the page(s) of this web document that contain the proof, and make changes by hand on that printout.
5. Consider the following π-calculus model.
   $$!\ a(v).\overline{v}\langle \mathrm{p}()\rangle.0 \quad | \quad !\ (\nu c)\overline{a}\langle c\rangle.c(y).\mathrm{q}(y)$$
   Here, the notations p() and q(*x*) represent *sequential* computer functions, and are not part of the π-calculus notation.
   The function p() requires no arguments and sequentially produces a return value (output) when called (applied).
   The function q(*x*) requires one argument (input) *x* and performs some sequential operation with that argument when called.
   Answer the following questions:
   a. This model formally describes an interaction between two programs running in parallel. Give an informal verbal description of what those two programs do and how they interact, according to the π-calculus expression above.
   b. Perform π-calculus reduction and structural congruence to work through one interaction between these two programs.
   c. You may give a thorough formal computation as in the proof of the incr RPC system, or you may skip or combine steps you feel comfortable with, as long as your work is accurate and expresses the calculation clearly.
6. Write your own π-calculus expressions for modeling each of the following parallel computations. (Each itemized sentence describes a separate problem to solve.) *Note:* No π-calculus replication operations are necessary for these problems, although you may optionally include it.
   a. One program uses channel *a* to send an integer value 5 and a new channel to another program, and that latter program sends twice that integer value back to the first program along that new channel.
   b. One program uses channel *b* to send an integer value 10 and a new channel to another program; that second program uses channel *c* to send twice that integer value and that same new channel to a third program; and that third program outputs three times the integer it receives along the channel it receives to the first program.

Examples and applications

- An exercise: Reduce the following π-calculus expression:
  $$(\nu x)\big(\overline{x}\langle z\rangle.0 \mid x(y).\overline{y}\langle x\rangle.x(y).0\big) \quad | \quad z(v).\overline{v}\langle v\rangle.0$$
- Mobile communications [Milner 91]

History; other formalisms

[Wing 02]

- The π-calculus is an example of a *process calculus*, i.e., a mathematical structure with a set of values and operations on those values in which processes are among the values and parallel composition ("running processes in parallel") is a commutative and associative operation on processes.
- The π-calculus was created in 1992 by Robin Milner, Joachim Parrow, and David Walker. Milner (1934-2010) was a famous British computer scientist known for inventing one of the early systems for automatic theorem proving (LCF) and for creating the functional programming language ML, in addition to the π-calculus.
- The π-calculus extends Milner's earlier process calculus system called CCS (Calculus of Concurrent Systems) [Milner 80].
- Another famous British computer scientist, Tony Hoare, created a simlar system called CSP (Communicating Sequential Processes) [Hoare 85], starting about 1978. CSP is the theoretical basis for the Occam language for parallel programming.
- A major distinction between the π-calculus and predecessor systems is the π-calculus's ability to pass a channels from one process to another, along some other communication channel. This feature enables the system to model mobility, (e.g., cell phones) and changes in process structure.