Parallel and Distributed Compilations in Loosely-Coupled Systems: A Case Study

Erik H. Baalbergen

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands

ABSTRACT

One application of large-grain parallelism is the use of parallel and distributed compilations by *make*, running under UNIX.† The original version of *make* executes its compilation commands successively. An increase in efficiency may be achieved by a parallel version of *make*, which tries to execute the compilations simultaneously. A parallel, non-distributed, version of *make* turns out to be inefficient. The compilations, which are mainly cpu-bound, slow each other down due to degradation of the processor's performance. A solution may be found in the idea of boarding out (part of the) compilations to other processors. This resulted in a study of how to do compilations in a distributed manner.

The aspect of having a system of loosely-coupled processors is an important issue in the field of distributed compilations. The relatively high cost of doing transactions (compared to local actions) in a loosely-coupled system makes the use of low-level inter-processor communication (e.g., the execution of system calls on another processor) inefficient. A UNIX network system like the *Amoeba Connection* turns out to be unsuitable for doing distributed compilations. It is shown that much overhead results from the communication between the system that contains the source code to be compiled and the system that does the compilation. Another possibility is to copy the source code to the other processor's data space, execute a local compilation on that processor and send the results back; this greatly reduces the communication overhead. The time needed to send the source to and receive the code from the remote processor is negligible compared with the overhead mentioned earlier.

In order to create a parallel and distributed *make*, I adapted the original 'make' program by adding a module for finding out which compilations can be executed in parallel, depending on the actions to be taken, the actions already finished, and the present files. Furthermore, I created various versions of the UNIX C compiler *cc* in order to perform some measurements.

† UNIX is a trademark of Bell Laboratories.

1. INTRODUCTION

The availability of networks of personal workstations has increased interest in parallel and distributed compilations. A decrease of the response time is the main motive for executing compilations in parallel on several processors. This paper is a description of an experiment set up to examine the feasibility of using more than one processor for doing compilations. The experiment took place in a UNIX environment and consisted of two parts: creating two different distributed versions of the C compiler *cc* and constructing a version of the UNIX tool *make* [Feldman78] that could run its compilations in parallel. The aspect of being tightly-coupled or loosely-coupled turns out to be an important issue in determining whether a specific network is suitable for doing distributed compilations.

The test configuration consisted of four VAX 11/750s, each of them running UNIX version 4.1 BSD. The machines are connected by a 10 Mbps token ring (Pronet). Pronet was made available to user programs by incorporating the **Amoeba** 3.0 [Tanenbaum81] communication primitives into UNIX [Renesse84].

A distributed compilation can be done in various ways. One possibility is to create a version of the compiler in which the system calls may be carried out remotely. This can be achieved by using a UNIX network system like the *Amoeba Connection,* similar to the *Newcastle Connection* [Brownbridge82]. The system combines the file systems of each of the connected machines by allowing access to files and execution of programs on other systems. The compiler runs remotely (i.e., on another processor) but each system call concerning the source code should be executed on the processor that runs the file system of the source code. Another possibility is to isolate components from the compiler and execute some of them remotely. One problem with remote compilations is that the compiling program should produce code for the local machine. Each of the connected machines should therefore have a compiler for each of the other machines. This is no problem if the connected machines are similar, as is the case in the test environment. Another problem arises in the second kind of compilation: libraries and included source code should be derived from the source machine. This, too, caused no problem in our test environment, as will be shown.

It must be said that many of the results depend strongly on our configuration, especially with regard to the communication overhead. Results from a common network operation, performed in the same configuration, are included to give an indication of the overhead.

2. AMOEBA AND THE AMOEBA CONNECTION

Amoeba is a distributed operating system developed at the Vrije Universiteit [Tanenbaum81]. The **Amoeba** communication primitives are described in [Tanenbaum84]. **Amoeba** uses a "request-reply" or "transaction" style of communication, in which the basic primitive is the client sending a request to a server and the server sending a reply back to the client. Such a pair of request and reply messages is henceforth called a *transaction*. The implementation of the primitives in UNIX created the possibility for two processes running on different systems to communicate by means of transactions. This has been exploited in various user programs such as copying files between various systems, remote execution of commands, sharing of resources and remote logging in [Renesse84]. An application of the transaction mechanism is a UNIX system-call server. A process on machine *A* can ask a system-call server on machine *B* to execute a system call, such as

open, **read**, or **write**. The strategy used to implement the remote system calls is to build an extra layer on the kernel. A program does not directly invoke the kernel but calls a stub routine which checks whether the command must be done locally or remotely. Local commands are passed directly to the kernel. Remote commands are passed to a system call server on the proper machine by doing a transaction with the system-call server. A great advantage of the use of this extra layer is that existing programs need not be rewritten or even recompiled. They only have to be relinked with a library of stub routines. The naming scheme for remote files (i.e., files on other UNIX systems), the system-call server and the stub-routine library together form the *Amoeba Connection.* The connection was found useful in our experiment, although the overhead was large.

The following table gives an indication of the speed of the connection in terms of response time, measured in seconds. Three versions of the UNIX file-copy command *cp* are compared: plain *cp*, able to copy files on the local machine only; *rcp*, the inter-machine file-copy program as described in [Renesse84]; and *fcp*, which is plain *cp* linked with the *Amoeba Connection* library. All measurements took place on lightly loaded machines.

number of bytes	cp	rcp		fcp	
	local	local	remote	local	remote
1	0.28	1.08	3.55	0.25	0.40
1024	0.28	1.20	3.60	0.25	0.43
10240	0.40	1.93	3.95	0.38	0.83

Table 1.

3. THE EXPERIMENT AND ITS RESULTS

3.1. A distributed compiler

The first phase of the experiment was to construct two distributed versions of *cc* which is the UNIX C compiler. *Cc* is a program that causes C source code to be passed through several compilation programs. The first step is done by the C preprocessor *cpp* which performs macro substitution, file inclusion and elimination of source code, depending on several user-specified, preprocess-time conditions. Next follows the compiler proper, *ccom*, which is a two-pass portable C compiler [Johnson79]. The assembly code generated by the two-pass compiler is translated to object code by the UNIX assembler *as*. The program *ld*, which is the UNIX link editor, finally combines the object programs, together with some libraries, into one program which may be executed. The first distributed version of the C compiler is based on the *Amoeba Connection.* The four compilation programs are relinked to allow files on other systems to be compiled. The resulting compiler, together with its driver *cc*, is installed on each of the connected machines. (Having multiple copies of the compiler is in fact an optimization with regards to our implementation of the *Amoeba Connection*; remote execution is allowed only if the program is situated in the file system belonging to the remote processor.) The remote compilation of a source file on **A** is now done by invoking *cc* on some other machine **B.** The overhead caused by the implementation of the *Amoeba Connection* is large. A transaction takes one system call on the local machine and two system calls on the remote machine.

Furthermore, a remote system call takes one system call on the remote machine. It turns out that if the compiler executes one remote system call, four system calls are needed to get the result. Communication overhead, too, plays an important role. Some results, compared to local execution, are listed in table 2.

The second distributed version of the compiler was created using the local versions of each of the four components. *Cpp* and *ld* run locally, because both programs are strongly file-system dependent. *Cpp* may include some standard files and *ld* may use some libraries from the local file system. *Ccom* and *as* are combined into a server which takes preprocessed C-source code as input and produces object code as output. The input and output (and compiler diagnostics) may be sent across the network using transactions. The compiler server can therefore run remotely, getting its input from the locally-executed preprocessor and sending its output to the local file system. The advantage of this approach is that the various components need not be recompiled and that the only network communication is limited to invoking the server and passing the input and output. Some results for response times (compared to plain *cc*) are listed below.

number of C source lines	local using cc	Amoeba Connection		Compiler server	
		local	remote	local	remote
20	3.27	4.05	21.45	3.07	5.58
200	9.17	9.82	30.57	9.07	11.33
2000	68.75	68.68	105.28	68.98	69.63

Table 2.

Table 2 shows that remote compilation using the compiler server gives a better response time than using the compiler, based on the Amoeba Connection.

3.2. Distributed and parallel *make*

The second part of the experiment was to apply the result of the first part to the UNIX program *make*. Many programs developed under UNIX consist of a set of C source files which have to be compiled. *Make* performs the compilations sequentially. Doing the compilations in parallel on the same processor is not a solution, as table 3 shows. A solution is found in parallel and distributed execution of the compilations. Table 3 shows results of the three ways of doing independent compilations (i.e., no compilation uses results of the other compilations.) The file to be compiled was about 1800 lines. The maximum number of available processors was 4. The compiler server is used for doing the distributed compilations.

number of compilations	locally sequentially	locally in parallel	distributed in parallel
1	68.67	-	-
2	137.34	127.32	69.58
3	206.00	183.45	73.22
4	274.66	242.50	78.35

Table 3.

Adapting *make* in order to execute compilations in parallel on several processors was done easily. *Make* maintains a list of programs that can also run on other machines. A command is executed in parallel to the other commands under the following conditions: the program appears on the distributed-program list; the necessary files are present; the compilations on which the command depends are already done; and there is still enough remote processing power. The only program currently in the distributed-program list is *cc*, but other compilers and translating programs may be added to this list. The processing power of a machine is computed by keeping an account of the number of compilations started by *make* on that machine.

4. PLANS

From the experiment we learned that splitting up compilations and running the environment and machine independent phases on other processors results in a remarkable increase in response time, especially if several sources need to be compiled. Splitting up a single compilation is also a result of the philosophy behind the **Amsterdam Compiler Kit (ACK),** a project at the Vrije Universiteit in the area of compiler construction [Tanenbaum83]. A compilation in **ACK** causes a program to pass through several components: a front end, which translates the program into machine-independent intermediate code; a peephole and global optimizer; a back end, which translates from intermediate code into assembly code; an assembler; and a linker. The idea is to have a pool of processors, each of them running a dedicated server performing one of the components. **ACK** tries to allocate a server for each of the phases and passes the program through the pipeline of servers.

5. References

[Brownbridge82]

D.R. Brownbridge, L.F. Marshall and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!," *Software-Practice and Experience,* vol. 12, pp. 1147-1162 (1982).

[Devarakonda85]

M. Devarakonda, R. McGrath, R. Campbell and W. Kubitz, "Networking a Large Number of Workstations Using UNIX United," *Proc. 1st IEEE Int. Conf. on Computer Workstations,* pp. 231-239 (1985).

[Feldman78]

S.I. Feldman, "Make - A Program for Maintaining Computer Programs," appeared in *UNIX Programmers Manual,* vol. 2A, Bell Laboratories, Murray Hill NJ, 1979.

[Johnson79]

S.C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Sym. on Principles of Programming Languages,* pp. 97-104 (January 1978).

[Miller82]

J.A. Miller and R.J. LeBlanc, "Distributed Compilation: A Case Study," *Proc. 3th IEEE Int. Conf. on Distributed Computing Systems,* pp. 548-553 (October 1982).

[Renesse84]

R. van Renesse, A.S. Tanenbaum and S.J. Mullender, "Connecting UNIX systems Using a Token Ring," Rapport IR-91, Vrije Universiteit, Amsterdam, The Netherlands (October 1984).

[Tanenbaum81]
A.S. Tanenbaum and S.J. Mullender, "An Overview of the Amoeba Distributed Operating system," *Operating Systems Review,* vol. 15, no. 3, pp. 51-64 (July 1981).

[Tanenbaum83]
A.S. Tanenbaum, J.M. van Staveren, E.G. Keizer and J.W. Stevenson, "A Practical Toolkit for Making Portable Compilers," *CACM* vol. 26, no. 9, pp. 654-660 (September 1983).

[Tanenbaum84]
A.S. Tanenbaum and S.J. Mullender, "The Design of a Capability-Based Distributed Operating System," Rapport nr. IR-88, Vrije Universiteit, Amsterdam, The Netherlands (November 1984).