Transformation of Parallel Programs Guided by Micro-Analysis

Aline Weitzman
Brandeis University

October 5, 1992

[summary by Paul Zimmermann]

Abstract

In this talk A. Weitzman provides a summary of the past work done at Brandeis in micro-analysis, and outlines some new research directions which are the subject of her dissertation. The overall goal is to develop program manipulation and analysis tools which help a user in transforming parallel programs so as to render them more efficient for execution in a variety of parallel machines. One of the original contributions of this work is in the usage of symbolic processing and constraint logic programming to analyse and manipulate parallel programs for parallel computers. In the talk A. Weitzman provides: 1. a description of micro-analysis applicable to sequential programs, 2. micro-analysis approach for SIMD programs, 3. micro-analysis approach for MIMD programs, 4. automatic transformation scheme for translating SIMD programs to MIMD programs, guided by micro-analysis.

1. Introduction

Two different approaches are possible in order to establish the performance of a computer program: either make some *benchmarks*, that is measure the time used by the program on some input data, or try to derive some general formulæ (*time-formulæ*) that express the execution time of the program in terms of the time to perform basic (elementary) instructions, such as an addition of two registers, or a comparison between two variables. The second approach, which is called *micro-analysis*, has two main advantages. First it does not depend on a specific machine, because the time-formulæ are valid for any machine; only times to perform each basic instruction have to be determined for a given computer, but only *once* for each machine. Secondly the time-formulæ can be derived automatically, without running the program. The micro-analysis approach enables us to estimate the performance of programs on machines yet to be built, or to evaluate the effect of program changes.

Thus, the aim of micro-analysis is to derive from a program a *time-formula*, which is a symbolic formula for the execution time of the program. The symbolic variables of the time-formula are the *time-variables*, which represent the time to perform common elementary operations (addition, assignment, subscripting, loop overhead, ...). For example, the statement

```
a[i,k] := b*(c+d+e)
```

has the time-formula $t_{subs2} + t_{assign} + t_{mult} + 2t_{add}$, and the loop

```
for i := 1 to n do
   a[i,k] := b*(c+d+e)
```

has the time-formula $n(t_{foroh} + t_{subs2} + t_{assign} + t_{mult} + 2t_{add})$.

2. Analysis of sequential programs

The heart of micro-analysis is the *time-formula generator*. It takes as input the program to be analyzed, and interactively asks the user to provide

- the probabilities with which branches of conditionals (*if then else*) are executed. These probabilities can be fixed (numeric or symbolic values) or can be specified as functions of given parameters;
- the number of times **while**-loops are performed. This number may also be symbolic or a function of given parameters.

For example, with the following program as input,

```
z=0; j=1;
while (j<=n) {
   if (x[j]==1) z=z+y;
   j=j+1; y=2*y;
}
```

the time-formula generator asks for the probability that `x[j]==1` becomes true and the number of times the `while` loop is executed, the user answer respectively p and n. Then the system outputs the following time-formula, with the help of MAPLE:

$$Formula = n(t_{whileoh} + t_{lesseq} + t_{mult} + (p+2)t_{assign} + (p+1)t_{add} + t_{cond} + t_{subs1} + t_{equal}) + 2t_{assign}.$$

Determining loop invariants. In some cases, the number of times loops are performed can be determined automatically. This is done by the *finite-difference equations generator*. The idea is to produce a set of finite-difference equations for the variables that are of interest in a given loop. Then, using the termination condition of the loop together with initial conditions provided by the program or the user, one uses a symbolic algebra system like MAPLE to solve the difference equations. Consider for example the loop

```
i=a; j=b;
while (i+j<n) {
   ...
   j=2*i; i=i+c;
}
```

The finite-difference equations generator outputs the solution

```
i(zz) = a + c * zz, j(zz) = 2 * a + 2 * c * zz - 2 * c
```

where zz is the number of times the loop has been executed (0 at the beginning). This result gives the number of iterations of the loop:

$$N = \left\lceil \frac{n - 3a + 2c}{3c} \right\rceil .$$

Determining values of time-variables. Once the time-formula of a given program has been derived, to obtain an estimation of the time complexity for a particular machine, one has to determine the numeric values of the time-variables $(t_{add}, t_{mult}, \dots)$ corresponding to this machine. For this aim, the first method that comes to mind is to execute N times an elementary instruction op, for example in a *for* loop, to measure the time used T, and to use the approximation

$$t_{op} \sim \frac{T}{N}.$$

A more accurate method has been proposed by T. Hickey. The idea is to use a set of *benchmark* programs $P_1, \dots, P_m$. One first runs all programs on the machine with counters for each elementary instruction. One thus obtains a set of vectors $c_1, \dots, c_m$. Each vector c_j contains the values of the elementary instruction counters for the benchmark P_j. One runs once more every benchmark, but this time without the counters, and one measures the time t_j of the program P_j. Then one sets the following system of equations

$$(t_{foroh}, t_{add}, t_{mult}, \dots) \cdot c_j = t_j(1 + \epsilon_j), \quad 1 \le j \le m,$$

and one approximates the time-variables by the numeric values that minimize the sum

$$\sum_{1}^{m} |\epsilon_j|.$$

With this method, the author was able to determine the values of twenty different time-variables for two different machines (HP 9000 and Sun 3), with an error of at most 4%.

3. Analysis of parallel programs for SIMD machines

A SIMD (Single Instruction Multiple Data) machine is a parallel architecture where each processor has its own memory. The same program is executed synchronously on each processor, thus there is an instance of each variable of the program in every processor (they are called *parallel variables*). The processors communicate to each other with *send* or *get* instructions. In what concerns micro-analysis, the situation is very similar to sequential programs, except one has to consider the cost of communication instructions.

On the Connection Machine (CM-2) for example, there are two kinds of communications:

- grid communication: every processor, say number j, sends a value to a processor at a fixed distance d, thus to processor $j+d$. Experiments made by the author show that the cost of grid communication depends on the decomposition of the distance d into a sum (or difference) of powers of two. For example, $T_{\text{grid}}(42) = T_{\text{grid}}(32) + T_{\text{grid}}(8) + T_{\text{grid}}(2)$, and $T_{\text{grid}}(60) = T_{\text{grid}}(64) + T_{\text{grid}}(4)$. The lowest time is about $500\mu s$ on the CM-2;
- general communication: any processor j is allowed to send any value to a processor at any distance d_j. The cost of general communication is constant by hardware considerations: it is about $1500\mu s$ on the CM-2.

Thus the micro-analysis of SIMD programs is essentially the same as for sequential programs, except one has to introduce some new time-variables like t_{send}, t_{scan} to scan processors where a parallel variable has a given value, t_{pcoord} to get the processor number, t_{pvar_read} to read a parallel variable.

4. Analysis of parallel programs for MIMD machines

A MIMD (Multiple Instruction Multiple Data) machine is a parallel architecture where each processor executes its *own* set of instructions. An example is the Butterfly computer, where the main program, executed on one processor, can start some new processes on other processors, and wait for their answer. This architecture is asynchronous, thus the time-formulæ are of the form

$$Time = T_a + \max(T_b, T_d) + T_c$$

where T_a, T_b, T_c, T_d are time-formulæ themselves. Thus it is possible to determine conditions on time-variables such that the maximum of T_b and T_d is T_b, and *vice-versa*. Finally, one obtains a time-formula of the form

$$Time = \begin{cases} f_1 & \text{if } c_1 \\ f_2 & \text{if } c_2 \\ & \vdots \\ f_k & \text{if } c_k \end{cases}$$

where the f_j are time-formulæ without any max function, and the c_j are sets of linear constraints involving the time-variables.

5. Transformation of SIMD into MIMD programs

The micro-analysis of SIMD and MIMD programs suggests to the author the following two-step algorithm to translate a SIMD program into a MIMD program:

- first translate the SIMD program into an intermediate sequential program, by replacing all parallel variables by arrays (the index represents the processor number), and regrouping instructions that do not need synchronization in blocks as large as possible.
- then translate the intermediate sequential program into a MIMD program, by dividing the iterations of synchronization-free loops between the MIMD processors. The optimal number of processors is computed using micro-analysis tools (because of the overhead for starting a new process, the optimal number is not necessarily the maximum).

Appendix: on the cost of grid communication for the CM-2

A. Weitzman found the following interpolation formulæ for the communication time $F(n)$ for distance n. F_i represents F on the interval $[2^i, 2^{i+1}]$. The function F has a recursive representation defined by the following formula:

$$\begin{aligned} F_5 &= -|18(n-48)| + 801 \quad \text{if } 2^5 \le n \le 2^6 \\ F_6 &= -|18(n-96)| + 1089 \quad \text{if } 2^6 \le n \le 2^7 \end{aligned}$$

$$F_i = \left\{ \begin{array}{ll} \Delta & \text{if } 2^i \le n \le 2^i + 2^5 \\ \Delta' & \text{if } 2^{i+1} - 2^5 \le n \le 2^{i+1} \\ F_k + 576 & \text{if } \left(\begin{array}{l} 2^i + 2^k \le n \le 2^i + 2^{k+1} \quad \text{or} \\ 2^{i+1} - 2^{k+1} \le n \le 2^{i+1} - 2^k, \\ \text{where } 5 \le k \le i-2 \end{array} \right) \end{array} \right\} \quad \text{if } 2^i \le n \le 2^{i+1},\ i \ge 7$$

where :

$$\begin{aligned} \Delta &= 18(n+28) \\ \Delta' &= 18(-n+28) \end{aligned}$$

For example, F_8 is $\Delta \diamond F_5 \diamond F_6 \diamond F_6 \diamond F_5 \diamond \Delta'$, where $f \diamond g$ represents the juxtaposition of f and g. In fact, F_i can be represented as $\Delta \diamond F_5 \diamond F_6 \diamond \cdots \diamond F_{i-2} \diamond F_{i-2} \diamond \cdots \diamond F_6 \diamond F_5 \diamond \Delta'$. The function F_i can be approximated using the central limit theorem:

$$F_i \approx 172i - 146 \quad \text{if } 2^i \le n \le 2^{i+1},\ i \ge 5$$

We can define the optimal cost $F(n)$ of grid communication between two processors at distance n on the Connection Machine as follows:

$$\begin{aligned} F(0) &= 0 \\ F(2^k) &= 1 \\ F(n) &= 1 + \min(F(n-2^k), F(2^{k+1} - n)) \quad \text{for } 2^k < n < 2^{k+1} \end{aligned}$$

This function can be easily defined in MAPLE:

```
F := proc(n)
local k;
    k := log2(n);
    if n=2^k then 1 else 1+min(F(n-2^k),F(2^(k+1)-n)) fi;
end:

log2 := proc(n) if n=1 then 0 else 1+log2(iquo(n,2)) fi end:

> seq(F(i),i=1..50);

 1, 1, 2, 1, 2, 2, 2, 1, 2, 2, 3, 2, 3, 2, 2, 1, 2, 2, 3, 2, 3, 3, 3, 2, 3, 3,

     3, 2, 3, 2, 2, 1, 2, 2, 3, 2, 3, 3, 3, 2, 3, 3, 4, 3, 4, 3, 3, 2, 3, 3
```

where `iquo(n,2)` is the MAPLE notation for the integer quotient of n by 2, and the auxiliary function `log2` computes the floor of the logarithm in base 2. The smallest n such that $F(n) = 4$ is $43 = 2^5 + 2^3 + 2^1 + 2^0$.

This function F appears in several fields of computer science. For example, in arithmetics and number theory, $F(n)$ is the minimal number of multiplications or divisions needed to compute a^n, once we know $a, a^2, a^4, a^8, \ldots$. This is the well-known problem of addition-subtraction chains which was studied by F. Morain and J. Olivos [2] to speed up the computations on an elliptic curve.

The sequence $(f_n = F(n))$ is also interesting because it is 2-regular, that is that the sub-sequences (f_n), (f_{2n}), (f_{2n+1}), (f_{4n}), (f_{4n+1}), (f_{4n+2}), (f_{4n+3}), $(f_{8n}), \ldots$ span a vector space of finite dimension. Namely, Ph. Dumas determined that $(f_n, f_{2n+1}, f_{4n+1}, f_{4n+3})$ is a basis of that vector space and that

$$f_{2n} = f_n, f_{8n+1} = f_{4n+1}, f_{8n+3} = f_{8n+5} = -f_n + f_{2n+1} + f_{4n+1}, f_{8n+7} = f_{4n+3}.$$

Philippe Dumas suggests that defining $\delta_n = \Delta f_n = f_{n+1} - f_n$ and using the Mellin-Perron formula as in

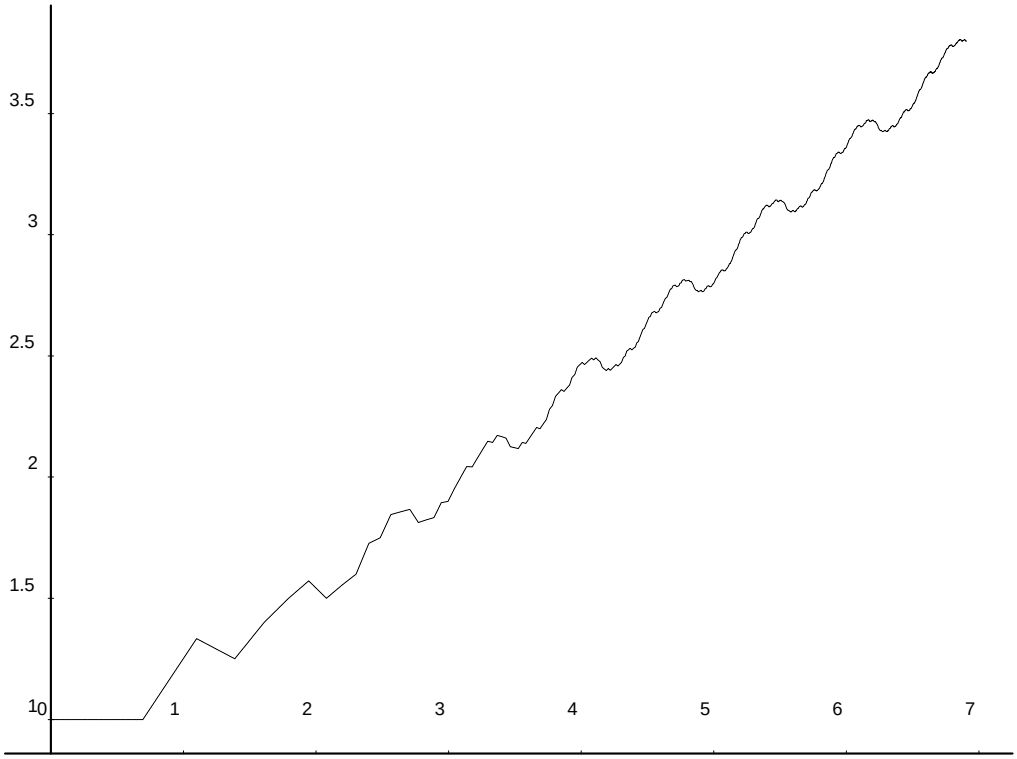

FIGURE 1. Plot of g_n/n as a function of $\log(n)$, for $1 \leq n \leq 1000$.

[1] would give some asymptotic estimates for the cumulated series $g_n = f_1 + \cdots + f_n$. In fact, Mordecai Golin already discovered a fractal form for g_n (figure 1) that looks like the fluctuations in the average case of Mergesort [1].

Bibliography

[1] Flajolet (Philippe) and Golin (Mordecai). – *Mellin Transforms and Asymptotics: The Mergesort Recurrence.* – Report, Institut National de Recherche en Informatique et en Automatique, January 1992. 11 pages.

[2] Morain (F.) and Olivos (J.). – Speeding up the computations on an elliptic curve using addition-subtraction chains. *RAIRO Technical Informatics and and Applications*, vol. 24, n° 6, 1990, pp. 531–543.