Bull. Nov. Comp. Center, Comp. Science, 31 (2010), 111–121
© 2010 NCC Publisher

Scalable parallel subdefinite calculations for sparse systems of constraints

Evgueni Petrov

Abstract. In the last 5 years, multi-core processors become more and more available to the wide public. Support for multi-core processors becomes de facto a standard for computation-intensive applications. In this paper, we present a parallel implementation of the UniCalc solver for non-linear engineering problems. Unlike the existent solvers of non-linear constraints, our parallel implementation scales well for sparse systems of non-linear constraints. We give scalability data for a quad-core processor.

1. Introduction

Constraint programming is a highly successful technology for solving the combinatorial problems (scheduling, staff allocation, assignment, routing, design, etc.) and non-linear constraints. Constraint programming toolkits are used by companies such as Amazon.com, British Airways, Chevron, Cisco, Ford, HP, KLM, Lockheed Martin, Nestle, Oracle, Proctor & Gamble, Renault, SNCF, UPS, and Volvo [1].

UniCalc is a constraint programming toolkit developed by the Russian Research Institute of Artificial Intelligence and A.P. Ershov Institute of Informatics Systems to answer the research requests from customers in industry, science and Russian government [12]. Most of such requests are related to reliable numerical solution of non-linear design and mathematical modeling problems.

The main constraint solver of UniCalc is called subdefinite calculations [9]. By their nature subdefinite calculations contain a big amount of parallelism. In this paper we present a parallel implementation of subdefinite calculations that scales well for dense and *sparse* systems of non-linear constraints. In experiments on a quad-core processor, our implementation demonstrated linear scalability on a number of benchmarks for non-linear constraint solvers over real numbers.

It is noteworthy that good scalability has been a challenge for some time in parallel constraint propagation over integral and real numbers. The existent implementations of parallel constraint propagation scale well only if the system of constraints is dense, i.e. each constraint involves a large number of variables. In such case, the time needed to process constraints dominates the time needed to update the queue of the constraints that wait to be processed, and good parallelization is relatively easy to achieve [13, 10, 2]. Probably for this reason, most recent papers on parallel constraint solvers over integral and real numbers focus on parallel exhaustive search in combination with sequential constraint propagation [11, 14, 15] rather than on parallel constraint propagation. Our major contribution is achieving good scalability in parallel constraint propagation in large *sparse* systems of non-linear constraints over real numbers.

The paper is organized as follows. Section 2 overviews the existent techniques of software parallelization. Section 3 describes subdefinite calculations. Section 4 outlines the parallel version of subdefinite calculations. Section 5 reports experimental performance scalability data for a set of benchmarks on a 4 core processor. Section 6 concludes the paper.

2. Software parallelization techniques

Support for multi-core processors becomes de facto a standard for computation-intensive applications. To provide that support, several libraries and language extensions have been developed. In this section we briefly review these means of parallelization from lower level to higher level: POSIX threads [4] and Windows threads libraries [5], Intel Threading Building Blocks library [6], OpenMP [7] and Cilk [8] language extensions. Distributed memory parallelism is out of our scope.

Ultimately, the libraries and extensions we discuss in this section are functionally equivalent, that is we can program any parallel behavior on the top of any of them. Below we point out the main differences between those libraries and language extensions.

POSIX threads and Windows threads provide threading and synchronization primitives that are very close to the level of the operating system and hence are as fast as one get. The application that uses such libraries is fully responsible for thread workload balance, thread data locality, and other higher level aspects of parallelization.

Intel Threading Building Blocks consist of a runtime resource manager and a C++ template library that provides higher level parallel constructs, such as parallel reduction, parallel for, blocked range, etc. The TBB runtime resource manager takes some care of balancing thread workloads by transferring workload from overloaded CPU cores to idle CPU cores. The overhead introduced by the TBB runtime resource manager is considered as reasonable for all available implementations.

The OpenMP and Cilk language extensions minimize the changes in the source code needed to parallelize an existent sequential application. OpenMP parallel constructs are introduced via compiler pragmas. Cilk parallel constructs are introduced via Cilk-specific keywords. Overall, Cilk is more advanced than OpenMP. In addition to parallel sections, parallel for, and parallel reduction, Cilk provides a more intelligent mechanism to balance thread workloads. The OpenMP and Cilk parallel constructs introduce smaller overhead than the TBB parallel constructs. OpenMP is more mature than Cilk. Only a few compilers support Cilk at the moment.

Our choice for parallelization is the OpenMP extension for C/C++ because we want to minimize the changes in the source code of UniCalc needed for parallelization.

3. Subdefinite calculations

Subdefinite calculations [9] have been invented by Alexander Narinjani in early 1990s. Subdefinite calculations are one of the first constraint programming techniques for constraint satisfaction problems with real numbers. In constraint programming terms, subdefinite calculations are constraint propagation for real intervals. Given a set of constraints and a bounding box for the set of solutions to the constraints, subdefinite calculations reduce this bounding box without losing any solution. Below we give a pseudo code for subdefinite calculations.

Let V be a set of variables that take values from the set $\mathbf{R}$ of real numbers. Let C be a set of constraints over V. Denote by proj(c, v) the convex hull of projection of the constraint c (as a subset of some $\mathbf{R}^k$) to the variable v (as an axis of the same $\mathbf{R}^k$). Denote by dependent(v) $\subseteq C$ the largest set of constraints that contain v. Denote by $c(v_1, v_2, \ldots, v_k)$ the fact that the constraint c contains only variables $v_1, v_2, \ldots, v_k$. Let Q be a set of constraints with the standard predicate empty and operations select (which can be non-deterministic) and union $\cup$. The set Q is called the set of active constraints. Finally, denote the Cartesian product by $\times$.

Figure 1 shows the algorithm of sequential subdefinite calculations. Notice that at compute time we use the variables V to denote projections of the bounding box to the axis of the solution space. A useful optimization for constraint propagation in inconsistent systems of constraints is to exit the while-loop as soon as *newvi* is empty (line 6 in Figure 1).

```
 1 Q = C;
 2 while (!empty(Q)) {
 3     c(v[1], v[2], ..., v[k]) = select(Q);
 4     Q = Q \ { c };
 5     for (i = 1; i <= k; k++) {
 6         newvi = proj(c ∩ v[1] × v[2] × ... × v[k], v[i]);
 7         if (v[i] != newvi) {
 8             Q = Q ∪ dependent(v[i]);
 9             v[i] = newvi;
10 }}}
```

Figure 1. Sequential subdefinite calculations

A floating point implementation of subdefinite calculations enjoys all the properties of constraint propagation: finite termination, preservation of the set of solutions to C, deterministic result (the bounding box for the set of solutions after the while-loop is exited) despite a non-deterministic operation select.

4. Parallel subdefinite calculations

By their nature, subdefinite calculations contain a large amount of parallelism because the bounding box for the set of solutions can be updated in parallel without damaging correctness of the algorithm (line 6 in Figure 1).

Because parallelization applies mainly to loops, the first attempt is to parallelize the while-loop (line 2 in Figure 2) and to protect the set of active constraints by either an OpenMP critical section or by an OpenMP lock (similar to critical sections, not shown).

```
 1 Q = C;
 2 while (!empty(Q)) {
 3    #pragma omp parallel shared(Q, C)
 4    {
 5        #pragma omp critical
 6        {
 7            c(v[1], v[2], ..., v[k]) = select(Q);
 8            Q = Q \  { c };
 9        }
10        for (i = 1; i <= k; k++) {
11            newvi = proj(c ∩ v[1] × v[2] × ... × v[k], v[i]);
12            if (v[i] != newvi) {
13                #pragma omp critical
14                {
15                    Q = Q ∪ dependent(v[i]);
16                }
17                v[i] = newvi;
18 }}}}
```

Figure 2. Parallel subdefinite calculations with limited scalability

Notice that we do not have to serialize updates to the bounding box (line 11 in Figure 2) because they are monotonic with respect to set inclusion. Though a CPU core ignores some updates to the bounding box made by its colleagues, this fact does not lead to a loss of precision since all CPU cores share the same set of constraints. The worst consequence might be a minor loss of performance.

The problem with this version is that it must serialize updates to the shared set of active constraints (lines 8 and 15 in Figure 2). This fact limits the gain from additional CPU cores for large systems of constraints, especially for large *sparse* systems that consist of simple constraints. Under such conditions, updates to the bounding box are faster than updates to the set Q and, by Amdahl's law, the scalability factor is limited by 2x for any number of CPU cores. This fact agrees well with the scalability data from [10].

To fix the problem in the parallel implementation of subdefinite calculations in Figure 2 and enable parallel updates to the set of active constraints, we run the entire algorithm of subdefinite calculations on each CPU core and let them share the bounding box.

However, a CPU core ignoring updates to the bounding box made by its colleague may exit the parallel section at the very beginning of constraint propagation and this fact will damage scalability considerably. To work around such a premature termination, each worker thread tracks locally without synchronization/serialization (line 14 in Figure 3a; the function width returns the width of intervals) whether it has any chances to update the bounding box. If there are such chances, it starts a new session of constraint propagation (line 15 in Figure 3a).

```
 1 #pragma omp parallel shared(C)
 2 {
 3      do {
 4         Q = C; solved = 1;
 5         while (!empty(Q)) {
 6             c(v[1], v[2], ..., v[k]) = select(Q);
 7             Q = Q \  { c };
 8             for (i = 1; i <= k; k++) {
 9                 newvi = proj(c ∩ v[1] × v[2] × ... × v[k], v[i]);
10                 if (v[i] != newvi) {
11                     Q = Q ∪ dependent(v[i]);
12                     v[i] = newvi;
13                 }
14                 solved *= width(v[i]) < ϵ;
15      }}} while (!solved);
16 }
```

Figure 3a. Scalable parallel subdefinite calculations (convergent case)

Implementation in Figure 3a needs yet another adjustment for the case where subdefinite calculations cannot locate the solution to the constraints C with the desired accuracy ϵ because the constraints have two (or more) solutions that do not fit into a bounding box of size ϵ, or the constraints are inconsistent, or simply not amenable to subdefinite calculations. The adjustment is to track the number of CPU cores that execute the while-loop (lines 5-14 in Figure 3a) and to exit the parallel section as soon as there are no such CPU cores. Without this adjustment, the algorithm in Figure 3a enters an infinite loop if the bounding box never gets smaller than ϵ in some dimension.

The adjusted algorithm is shown in Figure 3b. This adjusted algorithm may lose scalability on the systems of constraints that cannot be solved by subdefinite calculations with the desired accuracy. However, if subdefinite calculations converge to a small bounding box, the algorithms in Figure 3a and Figure 3b have the same scalability. In the next section, we show the scalability data for the parallel implementation of subdefinite calculations in Figure 3b. Good scalability is due to the fact that the while-loop (lines 8–18 in Figure 3b) "overweighs" the serialized increments/decrements to the counter *alive* (lines 6–7 and 19–20 in Figure 3b).

```
 1 alive = 0;
 2 #pragma omp parallel shared(C, alive)
 3 {
 4     do {
 5        Q = C; reduced = 0;
 6        #pragma omp atomic
 7            alive += 1;
 8        while (!empty(Q)) {
 9            c(v[1], v[2], ..., v[k]) = select(Q);
10            Q = Q \ { c };
11            for (i = 1; i <= k; k++) {
12                newvi = proj(c ∩ v[1] × v[2] × ... × v[k], v[i]);
13                if (v[i] != newvi) {
14                    Q = Q ∪ dependent(v[i]);
15                    v[i] = newvi;
16                    reduced = 1;
17                }
18        }}
19        #pragma omp atomic
20            alive -= 1;
21     } while (reduced && alive);
22 }
```

Figure 3b. Scalable parallel subdefinite calculations (general case)

5. Scalability of parallel subdefinite calculations

We experimented with non-linear constraint satisfaction problems from the public AMPL repository [3] translated to the UniCalc language. Each problem involves 50-5000 variables and constraints. Please see Appendix for details immediately, or visit the AMPL repository.

We benchmarked our parallel implementation of subdefinite calculations on a quad-core system at 3GHz and with 8G of RAM. The best of 8 times was recorded for each problem and a given number of threads. Each OpenMP thread was pinned to a unique CPU core.

Figure 4 gives a table with the time (in seconds) spent by our parallel implementation of subdefinite calculations to solve each constraint system to accuracy of 10^{-4} or higher, and a chart with the speedup compared to 1 CPU core. The second column in the table in Figure 4 shows how *sparse* is each constraint system – the number of variables in the biggest constraint and the total number of variables in the constraint system. We see that our implementation scales linearly for *sparse* and dense systems of non-linear constraints.

	#var (constr. / total)	1 core	2 cores	3 cores	4 cores
artif	3/5000	0.281	0.203	0.172	0.141
bdvalue	3/100	17.61	8.000	5.437	4.156
broydn3d	3/1000	2.625	0.641	0.422	0.297
cbratu2d	6/529	8.375	3.813	2.734	2.063
cbratu3d	9/2000	2.500	1.031	0.766	0.609
chandheq	64/64	0.188	0.078	0.063	0.047
chemrcta	5/200	47.38	20.06	13.72	9.547
chemrctb	2/100	9.671	5.203	3.078	2.391
integreq	127/127	3.969	1.984	1.469	1.203

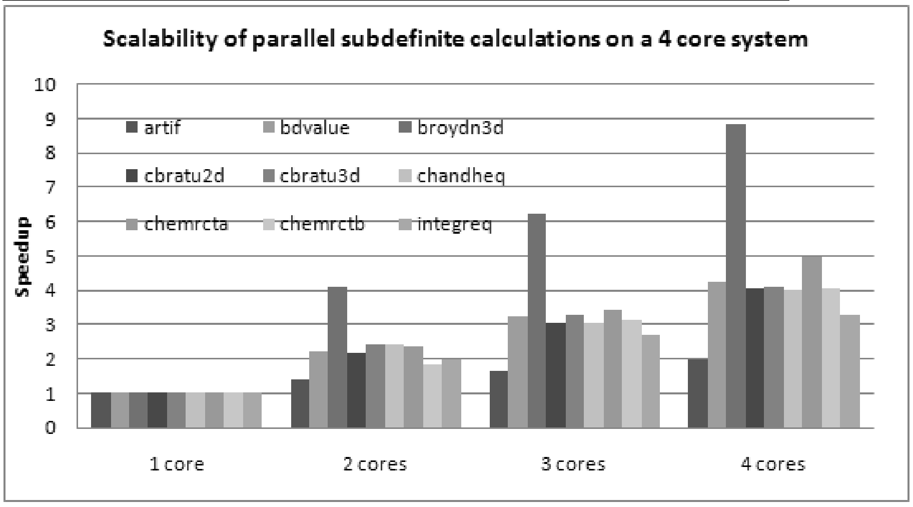

Figure 4. Scalability data for parallel subdefinite calculations

6. Concluding remarks

In this paper, we presented a parallel implementation of an interval constraint propagation algorithm called subdefinite calculations.

The value of our implementation of interval constraint propagation is that it scales well for large *sparse* (and this is the novelty of our work) and dense systems of non-linear constraints. Our contribution is that we give an experimental evidence that good scalability is feasible for constraint propagation in large *sparse* systems of non-linear constraints. The key feature of our implementation of the interval constraint propagation algorithm that enables its good scalability is a very limited amount of synchronization.

We plan to benchmark and tune our implementation of interval constraint propagation for systems with a large number of CPU core.

References

[1] van Beek P., Walsh T. Principles of Constraint Programming and Constraint Processing: A Review // AI Magazine. – 2004. – Vol. 25, No. 4 .

[2] Rolf C. Ch., Kuchcinski K. Parallel Consistency in Constraint Programming // // Proc. Internat. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA). – CSREA Press, 2009. – P. 638–644.

[3] Vanderbei R. Cute AMPL models. – Web site http://www.orfe.princeton.edu/˜rvdb/ampl/nlmodels/cute/

[4] The Open Group and IEEE. POSIX Threads // IEEE Standard 1003.1. – The Open Group and IEEE, 2004.

[5] Richter J., Nasarre Ch. Windows (R) via C/C++, 5th Edition. – Microsoft Press, 2007. – ISBN 9780735624245.

[6] Reinders J. Intel Threading Building Blocks. – O'Reilly Print, 2007. – 336 p. – ISBN 9780596514808.

[7] The OpenMP API specification for parallel programming. – Web site http://openmp.org

[8] Blumofe R. D., Joerg Ch. F., Kuszmaul B. C. et al. Cilk: An Efficient Multithreaded Runtime System // Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). – 1995. – P. 207–216.

[9] Narin'yani A. S. *Sub-Definiteness, Over-Definiteness, and Absurdity in Knowledge Representation (Some Algebraic Aspects) // Proc. Conf. Artificial Intelligence Applications 1985. – IEEE Computer Society/North-Holland, 1985. – P. 142–143.

[10] Granvilliers L., Hains G. A conservative scheme for parallel interval narrowing // J. Inf. Process. Lett. – 2000. – Vol. 74, N 3–4. – P.141–146.

[11] Beelitz Th., Bischof Ch. H., Lang B., Sch K. Althoff. Result-Verifying Solution of Nonlinear Systems in the Analysis of Chemical Processes // Lect. Notes Comput. Sci. – Springer Berlin, 2004. – Vol. 2991 – P. 198–205. – ISBN 3540212604.

[12] Botoeva E., Kostov Yu., Petrov E. A reliable linear constraint solver for the UniCalc system // Joint Bull. of NCC& IIS. Ser.: Comput. Sci. – 2006. – Iss. 24. – P. 101–111.

[13] Kasif S. On the parallel complexity of discrete relaxation in constraint satisfaction networks // J. Artif. Intel. – 1990. – Vol. 45, No. 3 – P. 99–118.

[14] Bordeaux L., Hamadi Y., Samulowitz H. Experiments with Massively Parallel Constraint Solving // Proc. Int. Joint Conf. on Artif. Intel. – 2009. – P. 443–448.

[15] Kalinnik N., Schubert T., Ábrahám E. et al. Picoso – A Parallel Interval Constraint Solver // Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA). – CSREA Press, 2009. – P. 473–479.

Appendix

The UniCalc language is derived from the conventional mathematical notation. Large systems of constraints are specified with the help of the **all** construct. For example, "**all**(i=1,2,5;x[i]*x[i+1] = i);" is equivalent to "x[1]*x[2] = 1; x[3]*x[4] = 3; x[5]*x[6] = 5;". Large sums and products are specified with the help of the **sum** and **prod** constructs. For example, "**sum**(i=1,2,5;x[i]*i)" is equivalent to "x[1]*1+x[3]*3+x[5]*5". For more detail on the UniCalc language, please, see [12].

artif

```
const int N = 5000;
all (i = 2, 1, N+1; x[i] >= 0.00001;
        (-0.05*(x[i] + x[i+1] + x[i-1]) + atg( sin( i*x[i] ) )) = 0);
x[1] = 0.0; x[N+2] = 0.0;
```

bdvalue

```
const int ndp = 100;
h=1/(ndp-1);
all(i = 2, 1, ndp-1; x[i] > -1;
        ( -x[i-1]+2*x[i]-x[i+1]+0.5*h^2*(x[i]+i*h+1)^3 ) = 0);
x[1] = 0.0; x[ndp] = 0.0;
```

broydn3d

```
const int N = 1000;
kappa1 = 2.0, kappa2 = 1.0;
all(i = 1, 1, N; x[i] = [-1,1]);
(-2*x[2]+kappa2+(3-kappa1*x[1])*x[1]) = 0;
all(i = 2, 1, N-1; (-x[i-1]-2*x[i+1]+kappa2+(3-kappa1*x[i])*x[i]) = 0);
(-x[N-1]+kappa2+(3-kappa1*x[N])*x[N]) = 0;
```

cbratu2d

```
const int p = 23;
lambda = 5.0; h = 1/(p-1); c = h^2/lambda;
all(i = 2, 1, p-1; all(j = 2, 1, p-1;
        (4*u[i,j]-u[i+1,j]-u[i-1,j]-u[i,j+1]-u[i,j-1]-
        c*exp(u[i,j])*cos(x[i,j])) = 0));
all(i = 2, 1, p-1; all(j = 2, 1, p-1;
        (4*x[i,j]-x[i+1,j]-x[i-1,j]-x[i,j+1]-x[i,j-1]-
                c*exp(u[i,j])*sin(x[i,j])) = 0));
all(j = 1, 1, p; all(i = 1, 1, p; u[i,j] = [-1,1]; x[i,j] = [-1,1]));
all(j = 1, 1, p; u[1,j] = 0.0; u[p,j] = 0.0; x[1,j] = 0.0; x[p,j] = 0.0);
all(i = 2, 1, p-1; u[i,p] = 0.0; u[i,1] = 0.0; x[i,p] = 0.0; x[i,1] = 0.0);
```

cbratu3d

```
const int p = 10;
lambda = 6.80812; h = 1/(p-1); c = h^2/lambda;
all(i = 2, 1, p-1; all( j = 2, 1, p-1; all(k = 2, 1, p-1;
        u[i,j,k] = [-1, 1]; x[i,j,k] = [-1, 1];
        (6*u[i,j,k]-u[i+1,j,k]-u[i-1,j,k]-u[i,j+1,k]-u[i,j-1,k]-u[i,j,k-1]-u[i,j,k+1]-
        c*exp(u[i,j,k])*cos(x[i,j,k])) = 0;
        (6*x[i,j,k]-x[i+1,j,k]-x[i-1,j,k]-x[i,j+1,k]-x[i,j-1,k]-u[i,j,k-1]-u[i,j,k+1]-
        c*exp(u[i,j,k])*sin(x[i,j,k])) = 0
)));
all(j = 1, 1, p; all(k = 1, 1, p;
        u[1,j,k] = 0.0; u[p,j,k] = 0.0; x[1,j,k] = 0.0; x[p,j,k] = 0.0
));
all(i = 2, 1, p-1; all( k = 1, 1, p;
        u[i,p,k] = 0.0; u[i,1,k] = 0.0; x[i,p,k] = 0.0; x[i,1,k] = 0.0
));
all(i = 2, 1, p-1; all(j = 2, 1, p-1;
        u[i,j,1] = 0.0; u[i,j,p] = 0.0; x[i,j,1] = 0.0; x[i,j,p] = 0.0
));
```

chandheq

```
const int n = 64;
c = 1;
all(i = 1, 1, n; x[i] = i/n; w[i] = 1/n; h[i] = [0, 1]);
all(i = 1, 1, n;
   sum (j = 1, 1, n; -0.5*c*w[j]/(x[i]+x[j])*h[j]*x[i]*h[i] + h[i]) = 1.0
);
```

chemrcta

```
const int n = 100;
pem = 1.0; peh = 5.0; d = 0.135; b = 0.5;
beta = 2.0; gamma = 25.0; h = 1/(n-1);
cu1 = -h*pem; cui1 = 1/(h^2*pem)+1/h; cui = -1/h - 2/(h^2*pem);
ct1 = -h*peh; cti1 = 1/(h^2*peh)+1/h; cti = -beta -1/h - 2/(h^2*peh);
all(i=1,1,n; t[i] = [0,1]; u[i] = [0,1]);
(cu1*u[2]-u[1]+h*pem) = 0; (ct1*t[2]-t[1]+h*peh) = 0;
all(i=2,1,n-1;
        (-d*u[i]*exp(gamma-gamma/t[i])+(cui1)*u[i-1] + cui*u[i] + u[i+1]/(h^2*pem))
        = 0;
        (b*d*u[i]*exp(gamma-gamma/t[i])+(cti1)*t[i-1] + cti*t[i] + t[i+1]/(h^2*peh))
        = 0
);
(u[n]-u[n-1]) = 0; (t[n]-t[n-1]) = 0;
```

chemctrb

```
const int n = 100;
pe = 5.0; d = 0.135; b = 0.5; gamma = 25.0;
h = 1/(n-1); ct1 = -h*pe; cti1 = 1/h + 1/(h^2*pe); cti = -1/h-2/(h^2*pe);
all(i = 1, 1, n; t[i] <= 1);
(ct1*t[2]-t[1]+h*pe) = 0; (t[n]-t[n-1]) = 0;
all(i = 2, 1, n-1;
   d*(b+1-t[i])*exp(gamma-gamma/t[i])+cti1*t[i-1]+cti*t[i]+t[i+1]/(h^2*pe)=0
);
```

integreq

```
const int N=127;
h = 1/(N+1); x[1] = 0; x[N+2] = 0;
all(i = 1, 1, N; t[i+1] = i*h; x[i+1] < 1);
all(i = 1, 1, N;
        ( x[i+1]+h*( (1-t[i+1])*sum (j = 1, 1, i; t[j+1]*(x[j+1]+t[j+1]+1)^3) +
        t[i+1]*sum(j = i+1, 1, N; (1-t[j+1])*(x[j+1]+t[j+1]+1)^3)/2) ) = 0
);
```