ON SOLVING LARGE SPARSE SYSTEMS OF NONLINEAR EQUATIONS USING THREADS

MIRCEA DRAGAN

ABSTRACT. This paper presents an algorithm for parallel solving of nonlinear systems. The current implementation uses threads on a Windows 2000/NT operating system and is independent on the numerical method used for solving the system.

1. INTRODUCTION

Solving large nonlinear systems of equations in parallel requires some additional tasks to be done. One of these tasks is to decompose the system into smaller subsystems in a proper way (see for example [5]). This task is not enough for solving the subsystems obtained. There is necessary a way to communicate among these subsystems solvers in order to get the proper solution. One possibility to do this is to use threads in Windows 2000/NT. This technique will be described in the paper. Similar work using threads can be found in [1].

Using threads is far from being the best method for parallel and distributed solving of subsystems obtained after decomposing the system. But this technique is, in some situations, the only one you can use, especially if you don't have a parallel computer or a network to solve the system using a better way.

In particular for our purpose the best way to use threads is to solve each subsystem obtained in a separate thread. One problem which arises is that we don't know in advance how many subsystems we have and, as a consequence, we don't know how many threads we need. The number of subsystems and the number of threads is determined at runtime and is dependent of the input data.

To solve the system in parallel we need to share a memory zone where we keep the solution. We also don't know in advance how big is the memory needed for solution and how we have to share data. All this information we get at runtime and thus the shared memory is dynamic allocated.

There are several methods how to write a multithreading program ([2], [3], [4], [6], [7]). A very important problem is how to deal with shared data. This can be done in several ways. One way is to use *critical sections*, which we use in our implementation.

AMS (MOS) *Subject Classification.* 68W10, 68W15, 65Y05, 65H10.

Key words and phrases. Parallel algorithm, asynchronous algorithm, iterative methods, sparse nonlinear system, thread, shared memory.

2. Description of the parallel method

In this section we present a method to write a parallel program using threads. The first step is to decompose the system into subsystems. This can be done, for example, using the algorithm below. A complete description of this algorithm can be found in [5].

```
proc decompose(G, s)
   empty the edge stack
   construct the adjacency structure of G
   for w a vertex do
      if w is not yet numbered then
         biconnect(w, 0)
      endif
      if edge stack is not empty then
         empty the edge stack and get a new subsystem
      endif
   endfor
   identify equations for each subsystem
   determine articulation points which belongs to two subsystems
end

proc biconnect(v, u)
   number [v] = + + i
   lowpt [v] = number [v]
   for w in the adjacency list of v do
      if w is not yet numbered then
         add (v, w) to edge stack
         biconnect(w, v)
         lowpt [v] = min (lowpt [v] , lowpt [w])
         if lowpt [w] ⩾ number [v] then
            start new subsystem
            while new top edge e = (u_1, u_2) on edge stack
                                 has number [u_1] ⩾ number [w] do
               delete (u_1, u_2) from edge stack
               if equation u_1 is not yet in current subsystem
                  add equation u_1 to current subsystem
               endif
               if equation u_2 is not yet in current subsystem
                  add equation u_2 to current subsystem
               endif
            enddo
            delete (v, w) from edge stack
            if equation v is not yet in current subsystem
               add equation v to current subsystem
            endif
```

```
            if equation w is not yet in current subsystem
               add equation w to current subsystem
            endif
         endif
      else if (lowpt[w] ⩾ number[v]) and (w ≠ u) then
            add (v, w) to edge stack
            lowpt[v] = min(lowpt[v], number[w])
      endif
   endfor
end
```

The next step is to organize threads, which means creating, calling and destroying threads. Each thread is responsible to solve one subsystem and to communicate with others.

Each thread computes a solution of its subsystem, if one can be calculated. When one thread ends, it communicates this by returning a proper value to signalize success or failure in finding a solution. If any thread fails, then the whole system fails. The algorithm for this is

```
proc thread
   for each subsystem i
      success[i] = start_thread(i)
   endfor
   determine if all subsystems returned success or failure and return the result
end
```

The procedure **start_thread** creates the thread, computes the solution of its subsystem if one can be computed and waits until it finishes, as follows:

```
proc start_thread(i)
   state = subsystem(i)
   wait until the thread is finished
   return state
end
```

The variable *state* returns *true* in case of success and $false$ in case of failure.

In solving a nonlinear system, one step (iteration) is usual of type $x^{k+1} = F\left(x^k\right)$, where x^{k+1} contains the value which is actually calculated in the step and x^k contains the old value. Each thread computes a step of its subsystem using a shared memory zone (in fact a shared array) which contains the next iteration x^{k+1} of the whole system. In the case one thread needs some variables which don't belong to its subsystem, it can take their values from this shared array. When next step is actually performed, the algorithm used for the computation must be modified as follows: before an element of x^{k+1} is computed, the access to its value is locked until the computation is done, and after that the lock is removed. This way we are sure that no other thread has access to this value during the

updating process. During the locking process all other threads which need access to this value are waiting until the lock is removed.

The values of the array x^{k+1} are actually computed in an asynchronous way, such that if one subsystem gives the next iteration faster than another, for the first subsystem the old values of the second subsystem are used.

The locking mechanism is implemented using critical sections ([4], [6], [7]). The algorithm which performs this is

proc subsystem(i)
  for $j = 1$ **to** *number of equations in subsystem i*
    perform *operations related to the solving method*
      which don't involve computation of x_j^{k+1}
    enter *critical section*
      compute x_j^{k+1}
    leave *critical section*
    perform *operations related to the solving method*
      which don't involve computation of x_j^{k+1}
  endwhile
  return *failure or success in getting a solution for the subsystem*
end

In the algorithm above we denoted by x_j^{k+1} the j^{th} component of x^{k+1} being calculated. The number of equations in each subsystem we know only at runtime.

In a numerical method used for solving the subsystems there might be some steps which do not involve any assignments to x_j^{k+1}. Such steps can be needed to be performed before the computation of x_j^{k+1} (which actually means using the old value for x_j^{k+1}) and some steps can be necessary to be done after the computation of x_j^{k+1} (which actually means using the new calculated value for x_j^{k+1}). The most important thing is that any assignment to x_j^{k+1} must be done in a critical section. In a concrete algorithm there can be more sections where a value for x_j^{k+1} is computed. All these sections must be critical, otherwise the behavior of the algorithm is not defined.

The algorithm is independent of the method used to solve the system.

3. Experimental results

In this section results obtained for different systems are illustrated. We compare the results obtained with the same method and input data, using the algorithm described above, and solving the system without decomposing it. The method chosen is Newton SOR and is not the best one, but can be used for our purpose.

One thing which should be mentioned is that there are some situations in which a system can be decomposed into subsystems and can be solved using the method above, but the system cannot be solve as a whole with the same method and input data, and the other way around.

To illustrate these situations, let's consider the system

$$\begin{cases} 2(x_1 - x_2) - 2\cos(x_3) - \sin(x_4 - 1) = 0 \\ \sin(x_1 - x_2)\frac{x_2}{2} - \sin(x_3) + \sin(x_4) = 0 \\ x_1 + x_3 - \tan(x_2) - x_4 = 0 \\ x_1 + x_5 - 4x_4 - 2 = 0 \\ 2(x_5 - x_6) - 2\cos(x_7) - \sin(x_8 - 1) = 0 \\ \sin(x_5 - x_6)\frac{x_6}{2} - \sin(x_7) + \sin(x_8) = 0 \\ x_7 - \tan(x_6) + x_5 - x_8 = 0 \\ x_5 + x_6 - 4x_8 + 2 = 0 \end{cases} \tag{3.1}$$

Using the algorithm from [5], the system (3.1) can be decomposed into three subsystems, $\{1, 2, 3, 4\}$, $\{4, 5\}$, respectively $\{5, 6, 7, 8\}$.

We denote by ω the relaxation factor for the Newton SOR method, by n the number of iterations and by ε the approximation error. We will illustrate some particular cases in solving this system in the tables below. We are interested in the number of iterations needed to get the solution with the approximation ε, out of a maximum of $20,000$ iterations. If we don't get the solution in $20,000$ iterations, then we will denote this entry in the tables by '*none*'. For the accuracy of solution we will consider for approximation error the values $\varepsilon = 10^{-5}$ and $\varepsilon = 10^{-8}$.

If we denote by x^0 the initial iteration, we consider the cases $x^0 = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $x^0 = \{4, 4, 4, 4, 4, 4, 4, 4\}$ and $x^0 = \{8, 7, 6, 5, 4, 3, 2, 1\}$. The number of iterations depending on ω is given in the following tables:

Table 1: Initial values $1, 2, 3, 4, 5, 6, 7, 8$

ω	Parallel		Normal	
	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$
0.4	372	612	372	612
0.5	175	274	182	275
0.6	120	168	96	149
0.7	431	207	135	165
0.8	*none*	*none*	*none*	*none*
0.9	60	73	69	82
1.0	177	294	*none*	*none*
1.1	393	442	544	563

Table 2: Initial values $4, 4, 4, 4, 4, 4, 4, 4$

ω	Parallel		Normal	
	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$
0.4	394	630	394	630
0.5	174	288	202	288
0.6	109	161	93	146
0.7	96	163	245	273
0.8	*none*	*none*	27	42
0.9	53	70	15763	15777
1.0	1306	8303	4687	4719
1.1	156	185	268	292

Table 3: Initial values $8, 7, 6, 5, 4, 3, 2, 1$

ω	Parallel		Normal	
	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$
0.4	366	601	397	620
0.5	161	257	189	280
0.6	108	158	108	158
0.7	211	292	302	333
0.8	*none*	*none*	*none*	*none*
0.9	123	183	135	148
1.0	*none*	*none*	886	1290
1.1	269	212	239	260

As one can see, there are two cases when the parallel method does not converge and the normal method converges (see Table 2 for $\omega = 0.8$ and Table 3 for $\omega = 1.0$). In this case the last subsystem (consisting of equations 5, 6, 7 and 8) has the same solution, but the first subsystem (consisting of equations 1, 2, 3 and 4) diverges. There are cases when one can compute the solution using only the parallel method (see Table 1 for $\omega = 1.0$). When both methods converge, the parallel method gives much faster the answer. This is obvious especially with large systems.

4. Conclusions

Using threads in solving a system in parallel is maybe not the best method, but sometimes this is the only one available, especially if we don't have a parallel computer or a network to solve the system using a better way.

The advantage of the algorithm proposed is that it is independent on the solving method used. The implementation can be also done on other operating systems which support multithreading.

In the algorithm proposed in this paper the subsystems obtained are solved in an asynchronous way. For some subsystems the next iteration can be computed much faster than for others, which can imply the instability of the solution. If we denote by n_i the

number of iterations for the subsystem i at a certain moment, we can introduce the relation that the difference between the maximum of n_i and the minimum of n_i to be less than a certain integer p.

As a future work the algorithm described in [5] will be extended using other techniques.

References

[1] B. Alkire, *Parallel Computation of Hessian Matrices under Microsoft Windows NT*, SIAM News, Vol. 31, Nr. 10.
[2] R. Asche, *Synchronization on the Fly*, Technical Report, MSDN Library, 1993.
[3] R. Asche, *Multithreading for Rookies*, Technical Report, MSDN Library, 1993.
[4] R. Asche, *Detecting Deadlocks in Multithreaded Win32 Applications*, Technical Report, MSDN Library, 1994.
[5] M. Dragan, *On Decomposing Large Sparse Systems of Nonlinear Equations*, Technical Report 01-05, RISC Linz, 2001.
[6] M. Pietrek, *Under the Hood*, Microsoft Systems Journal, 1996.
[7] J. Vert, *Writing Scalable Applications for Windows NT*, Technical Report, MSDN Library, 1995.

Software Competence Center, Softwarepark Hagenberg, Hauptstrasse 99, A-4232 Hagengerg, Austria

E-mail address: mircea.dragan@scch.at