COMMUNICATIONS IN NUMERICAL METHODS IN ENGINEERING
Commun. Numer. Meth. Engng, **14**, 773–781 (1998)

PARALLELIZING THE CONSTRUCTION OF INDIRECT ACCESS ARRAYS FOR SHARED-MEMORY MACHINES

JAROSŁAW TUSZYŃSKI AND RAINALD LÖHNER*
GMU/CSI, George Mason University, Fairfax, VA 22030-4444, U.S.A.

SUMMARY

A way has been found to form indirect addressing lists in parallel on shared-memory parallel machines. The maximum possible speed-up for typical tetrahedral grids is approximately 1:23. The algorithm requires an additional scratch array to shift from the serial 'elements surrounding points' to the parallel 'elements surrounding processors surrounding points' paradigm. The algorithm developed is general in nature, i.e. applicable to all indirect addressing lists. All numerical methods requiring the construction of indirect data structures, such as sparse matrix linear algebra procedures, field and particle solvers operating on unstructured grids, and network flow applications should see a benefit from this algorithm when running on shared-memory parallel machines. © 1998 John Wiley & Sons, Ltd.

KEY WORDS indirect address lists; unstructured grids; shared-memory parallel machines

1. INTRODUCTION

Indirect access lists, sometimes also referred to as Fortran linked lists, are often used to relate data of different types. Examples of this kind are:

(a) complementary data structures for unstructured grids, such as elements surrounding points, points surrounding points, edges surrounding points, etc.
(b) data structures for spatial search, such as points or particles belonging to buckets/octrees, faces covering spatial domains, etc.
(c) storage of interconnectivity data for graphs, e.g. for network flow applications[1]
(d) sparse matrix linear algebra procedures, e.g. skyline solvers.[2]

The storage of these data types may also be done using 2D arrays, an array of C-style linked lists or a Boolean adjacency matrix. However, for some applications, the most compact way of storing these data is via indirect access lists. Two 1D arrays are used to store what is, from a data-type viewpoint, 2D data. The first array stores the data in a contiguous way while the second array stores the storage locations for the items stored in the first one (see Figure 1). Given that

(a) arrays of this type need to be rebuilt often for adaptive unstructured grids, and
(b) shared memory machines with tens of processors, such as the SGI Power Challenge and SGI Origin, CRAY-T90, NEC-SX4, are becoming common,

* Correspondence to: R. Lohner, CSI, MS 4C7, George Mason University, Fairfax, VA 22030-4444, U.S.A.

CCC 1069–8299/98/080773–09$17·50

Received 18 June 1997
Accepted 15 January 1998

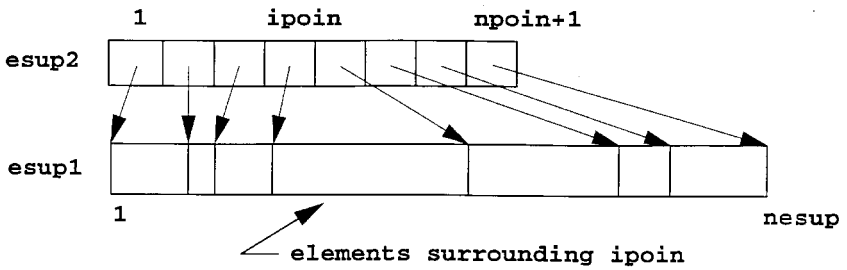

Figure 1. Definition of arrays esup1, esup2

the parallel construction of these indirect addressing lists has become important when trying to achieve fully scalable codes.

The remainder of the paper is organized as follows. Section 2 describes the serial code to construct the indirect addressing lists, and Section 3 treats the parallel construction. A theoretical speed-up estimate and measured timings are given in Section 4, and some conclusions are drawn in Section 5.

2. SERIAL CODE

In what follows, we illustrate all ideas, without loss of generality, on the elements surrounding the point list. Denoting by:

npoin: number of points
nelem: number of elements
nnode: number of nodes per element

the data structure describing the elements is given by the array of points belonging to each element lpoel(1 : nnode,1 : nelem). The task at hand is to derive, from lpoel, the list of elements surrounding points. With reference to Figure 1, we define:

nesup: the total number of elements surrounding points
esup1(1 : nesup): the list of elements surrounding points
esup2(1 : npoin + 1): the list of storage locations in esup1

The elements surrounding point ipoin are given by esup1(esup2(ipoin) + 1: esup2 (ipoin + 1)). The serial code builds these arrays by first counting the number of elements that surround a point and summing the storage locations needed. In a second pass over the elements the elements surrounding points are actually stored, and the final form of the esup2-array is obtained. The following is a Fortran implementation:

```
      npoi1 = npoin + 1
c - - - - loop 1: initialize esup 2
      do ipoin = 1, npoi1
        esup2(ipoin) = 0
      enddo
```



```
c - - - - loop 2: count the elements surrounding points
        do ielem = 1,nelem
          do inode = 1,nnode
            ipoin = lpoel(inode,ielem)
            esup2(ipoin) = esup2(ipoin) + 1
          enddo
        enddo
c - - - - loop 3: reshuffle esup2
        isto1 = 0
        do ipoin = 1,npoin
          isto0 = isto1
          isto1 = isto1 + esup2(ipoin)
          esup2(ipoin) = isto0
        enddo
c - - - - loop 4: store elements surrounding points in esup1, updating esup2
        do ielem = 1,nelem
          do inode = 1,nnode
            ipoin = lpoel(inode,ielem)
            istor = esup2(ipoin) + 1
            esup2(ipoin) = istor
            esup1(istor) = ielem
          enddo
        enddo
c - - - - loop 5: reorder esup2
        do ipoin = npoin,1, - 1
          esup2(ipoin + 1) = esup2(ipoin)
        enddo
          esup2(   1) = 0
```

Similar subroutines are used for other types of indirect access lists. The details may vary, but a two-pass approach in which indices are obtained in a first pass and the actual storage is performed in a second pass is common to the construction of most indirect access lists.

3. PARALLEL CODE

The serial code, although optimal in the number of operations and storage requirements, contains a number of recursions and possible memory contention that prevent a straightforward parallelization. These may be circumvented by introducing an additional temporary or scratch array `tstor(1:npoin,1:nprol)`, as shown in Figure 2. Here `nprol` denotes the number of independent processors which may be used for storage. Ideally, `nprol` should be the same as the number of processors `nproc`, but it is clear that `tstor` may become an impossibly large storage array for hundreds of processors. In the following, we proceed through loops 1–5 above, parallelizing each one in turn.

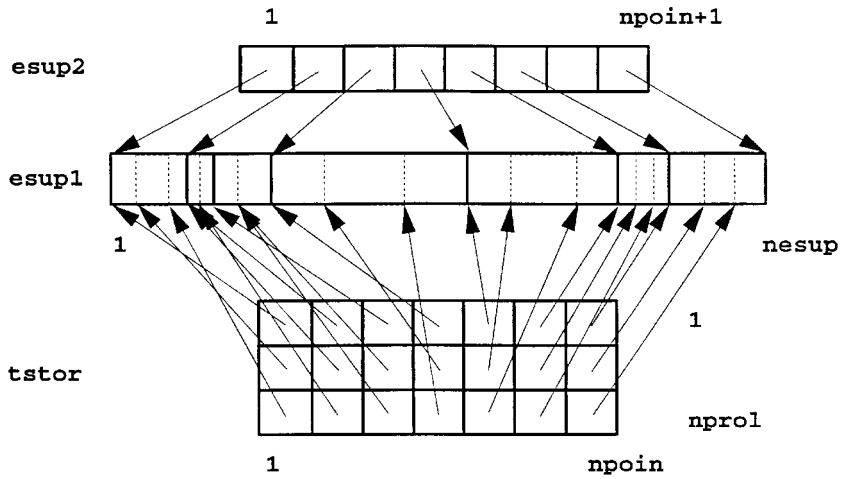

Figure 2. Relationship of scalar arrays to the parallel array tstor

3.1. Serial Loop 1

For the parallel case, the initialization of tstor replaces the initialization of esup2. This is a much larger array than before. If we assume that npoin $\gg$ nproc, the initialization may be carried out as

```
c - - - - loop 1p: initialize tstor to zero
        do iprol = 1,nprol
c$doacross local(ipoin)          ! compiler directive
          do ipoin = 1,npoin
            tstor(ipoin,iprol) = 0
          enddo
        enddo
```

Alternatively, one could have passed the complete array tstor as a 1D array of length nprol*npoin into a separate subroutine. This would have avoided the outer loop, yielding a longer parallel loop.

3.2. Serial Loop 2

From a parallel perspective, the main shortcoming of the serial loop is the possibility of memory contention when forming the indices in esup2. The idea is to work in groups or chunks of elements that are treated by each processor, storing the indices in the respective tstor locations:

```
c - - - - loop 2p.1: count the elements surrounding points
c - - - - obtain the number of elements treated by each processor
        nelel = (nelem + nprol-1)/nprol
c - - - - parallel loop over the processors
c$doacross local(iprol,iele1,iele2,ielem,ipoin)
        do iprol = 1,nprol
```



```
c - - - - range of elements covered by this processor
        iele1 = 1 + nelel * (iprol-1)
        iele2 = min(nelem,nelel * iprol)
c - - - - core of loop 2p.1: count the elements surrounding points
        do ielem = iele1,iele2
          do inode = 1,nnode
            ipoin = lpoel(inode,ielem)
            tstor(ipoin,iprol) = tstor(ipoin,iprol) + 1
          enddo
        enddo
      enddo
```

Having obtained tstor, one now has to build esup2, which is again done using the maximum number of processors available. At the same time, the entries in tstor are added in order to facilitate the parallelization of loop 4:

```
c - - - - loop 2p.2: form esup2 from tstor
c - - - - outer loop over nprol
        do iprol = 1,nprol
c - - - - parallel loops over all available processors
        if(iprol.eq.1) then
c$doacross local(ipoin)
        do ipoin = 1,npoin
          esup2(ipoin + 1) = tstor(ipoin,iprol)
        enddo
      else
        ipro1 = iprol-1
c$doacross local(ipoin)
        do ipoin = 1,npoin
         esup2(ipoin + 1) = esup2(ipoin + 1) + tstor(ipoin,iprol)
         tstor(ipoin,iprol) = tstor(ipoin,iprol) + tstor(ipoin,ipro1)
        enddo
      endif
      enddo
```

3.3. *Serial Loop 3*

Loop 3 may be parallelized by the standard two-pass strategy of recursions. The point-loop is divided into chunks of size npoin/nproc. In the first pass over the points, the indices of each chunk are obtained. A pass over the processors adds up the global indices. Finally, the proper index shift is added in a second pass over the points:

```
c - - - - chunk size (i.e. local number of points for each processor)
        npoil = (npoin + nproc - 1)/nproc
c - - - - loop 3p.1: first pass to reshuffle esup2
c$doacross local(iproc,ipoi1,ipoi2,ipoin)
        do iproc = 1,nproc
```



```
c - - - - extent of the points covered by this processor
          ipoi1 = 1 + npoil*(iproc - 1)
          ipoi2 = min(npoin,npoil*iproc)
c - - - - core of loop 3p.1
          do ipoin = ipoi1 + 2,ipoi2 + 1
            esup2(ipoin) = esup2(ipoin) + esup2(ipoin - 1)
            enddo
c - - - - store local sum
            lsump(iproc) = esup2(ipoi2 + 1)
          enddo
c - - - - loop 3p.2: add up local sums
          do iproc = 2,nproc
            lsump(iproc) = lsump(iproc) + lsump(iproc - 1)
          enddo
c - - - - loop 3p.3: second pass to reshuffle esup2
c$doacross local(iproc,ipoi1,ipoi2,iaddp,ipoin)
          do iproc = 2,nproc
c - - - - extent of the points covered by this processor
          ipoi1 = 1 + npoil*(iproc - 1)
          ipoi2 = min(npoin,npoil*iproc)
          iaddp = lsump(iproc - 1)
c - - - - core of loop 3p.3
            do ipoin = ipoi1 + 1,ipoi2 + 1
              esup2(ipoin) = esup2(ipoin) + iaddp
            enddo
          enddo
```

We remark that the number of operations in this loop is insignificant in comparison with the loops over elements for typical unstructured grids. For a moderate number of processors one can leave this loop as scalar without degradation of performance.

3.4. Serial Loop 4

Loop 4 is parallelized in the same way as loop 2. Groups or chunks of elements are treated by each processor independently. The storage into array esup1 is carried out without memory contention by referring to the tstor-array:

```
c - - - - loop 4p: store the elements surrounding points
          nelel = (nelem + nprol - 1)/nprol
c - - - - parallel loop over the processors
c$doacross local(iprol,iele1,iele2,ielem,ipoin)
          do iprol = 1,nprol
c - - - - range of elements covered by this processor
            iele1 = 1 + nelel*(iprol - 1)
            iele2 = min(nelem,nelel*iprol)
c - - - - core of loop 4p.1: store the elements surrounding points
            do ielem = iele1,iele2
```



```
      do inode = 1,nnode
        ipoin = lpoel(inode,ielem)
        istor = tstor(ipoin,iprol)
        esup1(esup2(ipoin) + istor) = ielem
        tstor(ipoin,iprol) = istor - 1
      enddo
    enddo
  enddo
```

Observe that tstor is essential in order to allow a paradigm shift from 'elements surrounding points' to 'elements surrounding processors surrounding points', enabling parallelization.

3.5. *Serial Loop 5*

Loop 5 is not required for the parallel version. tstor was used for internal index counts, leaving esup2 unchanged.

4. SPEED-UPS

The number of operations required for all loops involved are listed in Table I for the serial and parallel versions. These numbers assume that npoin $\gg$ nproc, and we denote nesup = nnode * nelem, r = nesup/npoin. This table only includes the actual number of integer or floating point operations, and would be indicative of advanced architectures and compilers with pre-fetch and post-store capabilities. Table II takes all operations into account.

Table I. Operation count (integer operations only)

Loop no.	Serial	Parallel	Maximum parallelism
1	npoin	nprol * npoin	nproc
2	nesup	nesup	nprol
		(3 * nprol-2) * npoin	nproc
3	npoin	2 * npoin	nproc
4	nesup	2 * nesup	nprol
5	npoin		
Total	(3 + 2r) * npoin	(4 * nprol + 3r) * npoin	

Table II. Operation count (Fetch,Ops,Store)

Loop no.	Serial	Parallel	Maximum parallelism
1	npoin	nprol * npoin	nproc
2	4 * nesup	4 * nesup	nprol
		(8 * nprol-5) * npoin	nproc
3	5 * npoin	7 * npoin	nproc
4	5 * nesup	7 * nesup	nprol
5	3 * npoin		
Total	(9 + 9r) * npoin	(9 * nprol + 2) * npoin	nproc
		+ 11 * r * npoin	nprol

If we denote $r = \texttt{nesup/npoin}$ and $t = \texttt{nproc/nrpol}$, then the theoretical speed-up S of the serial (s) vs. the parallel (p) code is therefore given by an expression of the form

$$S = nproc\left[\frac{\alpha_{s1}r + \alpha_{s2}}{\alpha_{p1}rt + \alpha_{p2}nprol + \alpha_{p3}}\right]$$

where α_{ij} are given from Tables I and II. The maximum theoretical speed-up for the case of infinite memory and very large number of processors is therefore

$$S_{\max} = \frac{\alpha_{s1}r + \alpha_{s2}}{\alpha_{p2}}$$

For an unstructured grid of tetrahedra we have $r \approx 22$, implying

$$12 < S_{\max} < 23$$

Figure 3 shows the bounding curves for a moderate number of processors. In order to verify these curves, timings were conducted on an unstructured grid of 2.2×10^6 tetrahedra with increasing numbers of processors. The machine used was an SGI Origin 2000 with eight processors, 4 Mbytes of cache and 4 Gbytes of memory. The speed-ups obtained are also displayed in Figure 3.

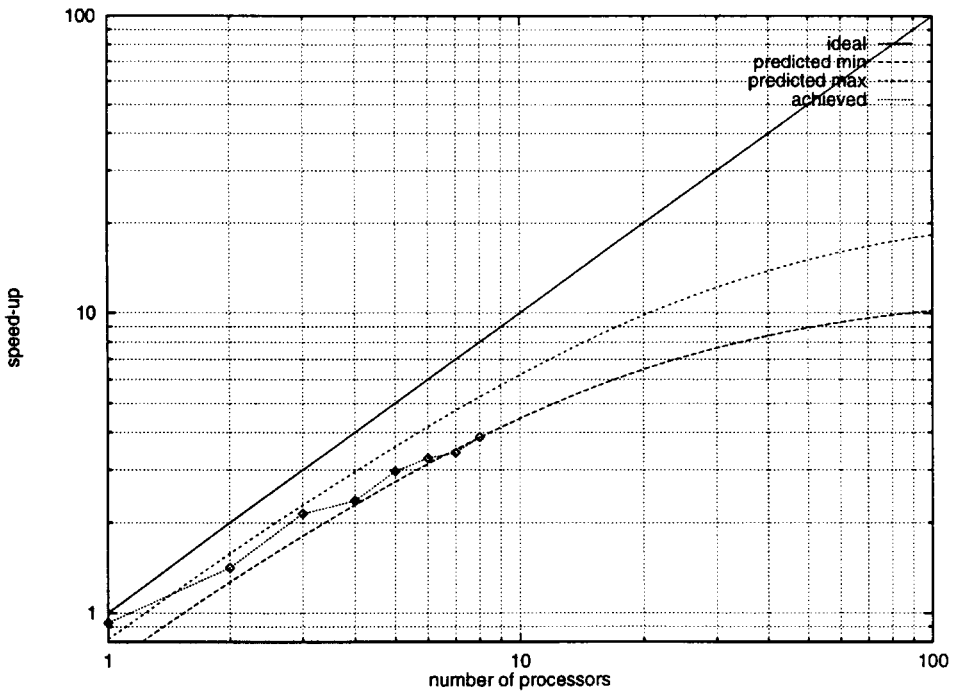

Figure 3. Theoretical and actual speed-ups

5. CONCLUSIONS

A way to form indirect addressing lists in parallel on shared-memory parallel machines has been found. The maximum possible speed-up for typical tetrahedral grids is approximately 1:23. The algorithm requires an additional scratch array to shift from the serial 'elements surrounding points' to the parallel 'elements surrounding processors surrounding points' paradigm. The array should ideally be of length `nproc*npoin`, which may appear impractical for very large numbers of processors. However, given that the maximal possible speed-up is bounded, memory requirements for near-optimal speed-up are bounded and are of order `20*npoin`.

The algorithm developed is general in nature, i.e. applicable to all indirect addressing lists. All numerical methods requiring the construction of indirect data structures, such as sparse matrix linear algebra procedures, field and particle solvers operating on unstructured grids, and network flow applications should see a benefit from this algorithm when running on shared-memory parallel machines.

ACKNOWLEDGEMENTS

This work was partially funded by AFOSR, with Dr Leonidas Sakell as the technical monitor.

REFERENCES

1. R. Sedgewick, *Algorithms in C*, Addison Wesley, 1990.
2. O. C. Zienkiewicz and R. Taylor, *The Finite Element Method*, McGraw Hill, 1988.

