Chapter 2
Principles of Shared Memory Parallel Programming Using *ParC*

This chapter introduces the basic concepts of parallel programming. It is based on the ParC language, which is an extension of the C programming language with block-oriented parallel constructs that allow the programmer to express fine-grain parallelism in a shared memory model. It can be used to express parallel algorithms, and it is also conducive for the parallelization of sequential C programs. The chapter covers several topics in shared memory programming. Each topic is presented with simple examples demonstrating its utility. The chapter supplies the basic tools and concepts needed to write parallel programs and covers these topics:

- Practical aspects of threads, the sequential "atoms" of parallel programs.
- Closed constructs to create parallelism.
- Possible bugs.
- The structure of the software environment that surrounds parallel programs.
- The extension of C scoping rules to support private variables and local memory accesses.
- The semantics of parallelism.
- The discrepancy between the limited number of physical processors and the much larger number of parallel threads used in a program.

2.1 Introduction

ParC was developed as part of a quest to make parallel programming simple namely easy to use and quick to learn. ParC is a block-oriented, shared memory, parallel version of the C programming language. ParC presents the programmer with a model of a machine in which there are many processors executing in parallel and having access to both private and shared variables.

The main feature of ParC is that the parallelism is incorporated in the block structure of the language. As such it makes the structure of a parallel program very similar to a sequential C program. This fully nested structure and the similarity to the structure of a sequential C program has the following implications:

Y. Ben-Asher, *Multicore Programming Using the ParC Language*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-4471-2164-0_2, © Springer-Verlag London 2012

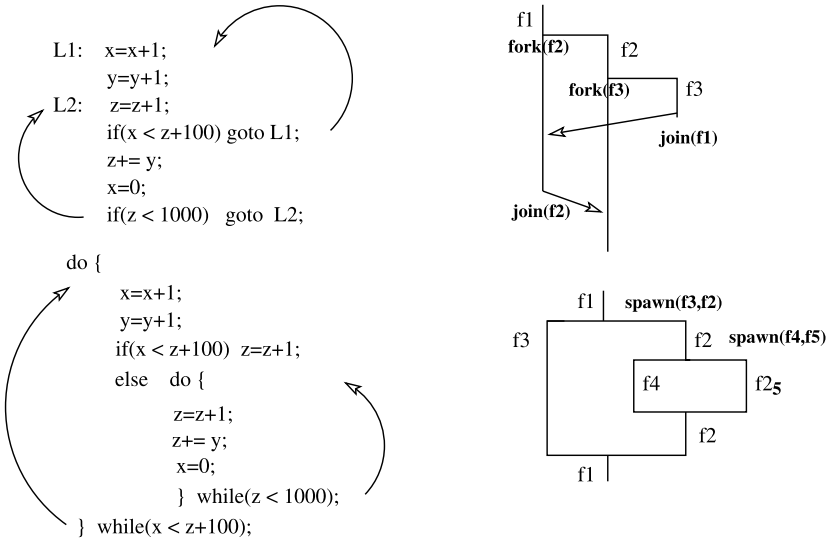


Fig. 2.1 General goto structure versus nested loops and general parallelism versus nested parallelism

- It leads to a nested structure of parallelism which, similar to sequential programming ease the task of coding parallel algorithms.
- It simplifies the task of transforming a parallel program to a sequential one and vice-versa. This is useful for debugging and gradual parallelization of sequential code (as will be discussed later).

ParC supports nesting of parallel constructs and as such it does not support arbitrary forms of parallelism. This is similar to the use of closed **while** loops instead of a general **goto** instruction. Figure 2.1 illustrates this concept showing a general goto structure versus the use of nested while-loops. This case is similar to the use of general *fork*(*thread*) and *join*(*thread*) operations that allow us to generate any graph of threads versus the use of nested parallelism. In nested parallelism a thread does not fork another thread but rather spawns children threads that must be merged eventually before the spawning thread can continue. Thus in the context of parallelism, nesting of constructs results in the use of closed parallel constructs rather than arbitrary **fork** and **join**. Similar to C's alternative loop-constructs (**for**, **do**, and **while**), ParC has several redundant parallel constructs, as well as redundant synchronization mechanisms.

ParC is accompanied by a virtual machine that virtually can execute or simulate ParC programs over a set of P processors. This virtual machine forms a model for executing ParC programs thus restricting the implementation of ParC to fulfill the guidelines of the virtual machine. Through this virtual machine and other assumptions, it is possible to estimate the execution time of a given program. The programmer can thus use this model to improve the performance of a given program. For

this purpose, ParC includes features and constructs to control the execution of a program by the virtual machine, e.g., mapping code-segments to specific processors at run time. Another direction to improve performances that is based on the ability to define execution time of parallel programs is to restrict the program to increase or decrease the amount of code that is executed by parallel constructs versus the amount of code that is executed sequentially. This tuning is usually done after the application has been fully developed and performance analysis has indicated the problem areas.

ParC is an extension of the *C* programming language, so there is no need to review any of its standard features.

2.1.1 System Structure

The environment of ParC is composed of three parts: a pre-compiler that translates ParC code into *C*, a simulator that simulates parallel execution on a UNIX machine, and a thread system library that supports the execution of ParC programs by a shared memory machine (SMM) (see Fig. 2.2).

The parallel program is represented by a single operating system process. Regular system calls may be used in a parallel program, however care must be given if they are used in parallel mode or to their effect on the execution. For example, executing a process-fork call in a ParC program may leave the child-process in a faulty non consistent state. All I/O operations must be serialized since file descriptors are shared by all the parallel threads spawned by the program. The subject of I/O operations during the run of a parallel program is more complicated, but for the time being it is assumed not to be part of the language but is subject instead to the way the underlying file system works. For example, assume that we have k threads that open the same file, store it in a local pointer and then write their IDs $1 \ldots k$ to the file and close it. It is unclear what the file will contain. Will it be one number or a permutation of the numbers $1 \ldots k$? This in general depends on the way in which the file system handles I/O operations. On multicore machines we may assume Posix I/O semantics for multi-threaded code. A more complicated question is, what happens if one of the threads closes the file or deletes it before other threads have completed their writing? These questions are outside the scope of this chapter but in general are related to practical parallel programming.

The pre-compiler is more complicated than a preprocessor compiling ParC's constructs to system calls for generating and executing the program's code segments in parallel. The pre-compiler has to recognize the language and keep a symbol table in order to support the scoping rules and other features. In particular, the compiler can generate new temporary variables and control the memory layout of the parallel program.

All these aspects are important when building a parallel program since they can affect its execution and performance.

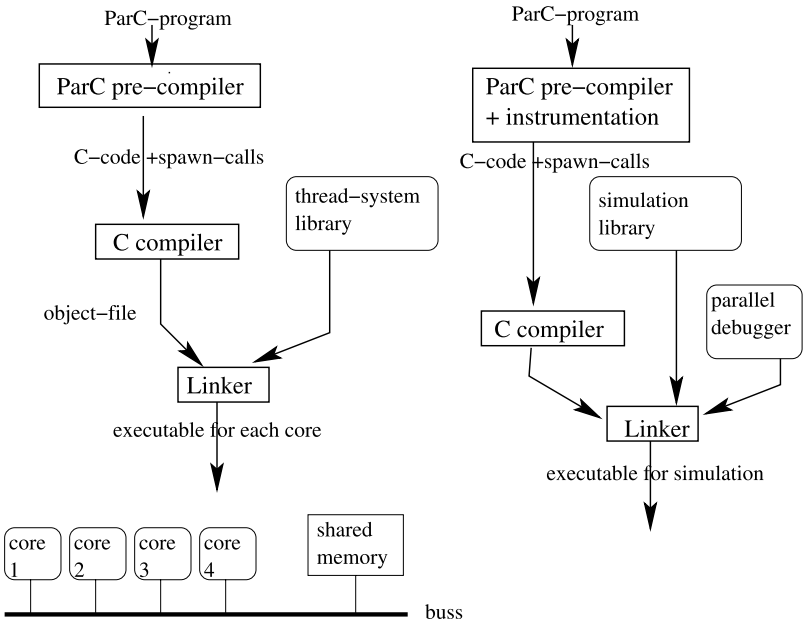


Fig. 2.2 The ParC software system

2.2 Parallel Constructs

Parallelism in parallel programs can be extracted from two sources of sequential constructs:

- loop's iterations that are executed in parallel and
- function calls and in particular recursion that are executed in parallel.

The parallel constructs in ParC are especially designed to handle both cases in an efficient and structured manner. The parallel "things" generated by ParC are called "*threads*." Each thread is a sequential code that may be executed in parallel to other threads (as explained in the previous chapter). Each thread has its own private stack that is allocated in the shared memory when the thread is generated. The thread stack allows function calls made by the threads to be executed separately from the function calls of other threads. The outcome of using "thread stacks" will be discussed later on.

Loop parallelism is obtained by the parfor construct, which is a parallel version of the conventional for loop in C. The syntax for this construct is shown on the right side of Fig. 2.3. The body of the parfor, `stmt`, can be any legal statement in C, including a function call, a loop or a conditional statement. If *stmt* is a block (compound statement), it can declare its own private variables, which can not be

Fig. 2.3 The parfor construct

```
parfor ( index = e1; index < e2; index+ = e3 )
{
      stmt
}
```

Fig. 2.4 Example of `parfor`

```
int array[100];
parfor(int i=0; i< 100; i++){
     array[i] = 0;
}
```

Fig. 2.5 Increasing the work done by each thread

```
int array[100000];
parfor(int i=0; i< 100000; i+=1000){
     int j;
     for(j=i; j<i+1000; j++) array[j] = 0;
}
```

referenced outside the scope of this parfor. In addition each iteration of the parfor use a different set of local variables (including the parfor's index) which are thus isolated from local variables of other iterations (unless passed by a reference through a global variable). This point is discussed in greater detail in Sect. 2.3. The bodies of the distinct iterates are executed in parallel. The meaning of the phrase "executed in parallel" is discussed in Sect. 3.8.

For example, the code segment in Fig. 2.4 may be used to initialize an array of 100 cells. The `parfor` generates 100 threads, indexed by the integer variable `i`. The index values ranges from 1 to 100 in increments of 1. Each thread comprises a single statement, which sets one cell of the array to zero. In case of a larger array we can let each thread to zero more than one element as depicted in Fig. 2.5. When using parallel constructs, one should keep in mind that spawning new threads incurs a certain overhead. This overhead can be around few hundreds of assembler instructions although for ParC threads it is usually smaller. Therefore, the threads should be substantial enough so that the benefits of parallelization justify the overhead. Threads with a single instruction, as in the example of Fig. 2.4 are definitely too small to be worthwhile hence code of the form depicted in Fig. 2.5 is preferable. We remark that care must be given to the index expressions that are used in parallel constructs. For example, the code in Fig. 2.5 is more efficient than the one depicted in Fig. 2.6.

A distinct, local copy of the loop index is created for each iterate containing the iteration-id similar to a regular sequential for-loop. The number of threads is

Fig. 2.6 A less efficient version

```
int array[100000];
parfor(int i=0; i< 100; i++;){
    int j;
    for(j=0; j<1000; j++) array[i*1000+j] = 0;
}
```

Fig. 2.7 Using `lparfor` to zero a large array

```
int array[100000];
parfor(int i=0; i< 100; i++){
    array[i] = 0;
}
```

$\lfloor \frac{e_2-e_1}{e_3} \rfloor + 1$ as in regular for-loops. The index variable in thread i is initialized to $e_1 + (i-1) \cdot e_3$. Note that this variable is only defined in the parfor and can not be referenced directly outside of its scope.

Unlike the C's $for(int\ i = e_1; i \leq e_2; i+ = e_3)stm;$, the expressions e_1, e_2, and e_3 are evaluated only once, before any threads are actually spawned. Thus, unlike regular *C*, changing the index inside a parfor construct will not affect its evaluation: This is done by using temporary variables to compute the loop's parameters before the execution of the loop begins:

```
parfor(i=0;i<n;i+=x)
  {
    A[i]++;
    if(A[i] > 7) i+=5;
    if(A[i] > 100) n+=5;
    if(A[i] > 5) x+=i;
  }
if(i < n) x++;
from=0; to=n; by=x;
parfor(index=from ;index< to;index+=by)
  { int i;
    i=index;
    A[i]++;
    if(A[i] > 7) i+=5;
    if(A[i] > 100) n+=5;
    if(A[i] > 5) x+=i;
  }
if(i < n) x++;
```

Thus, changing the parfor's iteration-range during execution does not have the same effect as changing these in regular for-loop nor is the loop's index affected in a similar manner.

```
int array[100000];
int np=100000/P; parfor(int ii=0; ii< 100000; ii+=np){
    int i;
    for(i=ii; i<ii+np; i++) array[i] = 0;
}
```

Fig. 2.8 Equivalent parfor version of an lparfor

Unlike C, it is illegal to transfer control via a Goto-statement or via setjump/long-jump either into or out of a parallel construct. Note that transferring control via goto is legal in *C*, as illustrated in the following example:

```
#include <stdio.h>
main(){
int i,j=4,n=100;
for(i=0;i<n;i++){
  j=j+1;
  l:  if(i == 10) goto t;
  else j = j+10;
}
for(i=0;i<n;i++){
  j=j+1;
  t:  if(i == 10) goto l;
  else j = j+10;
}
printf("%d %d\n",i,j);
}
```

which has no meaning had we used parfor instead of for-loops since the second parfor has not been started when the first goto occurs.

It is often the case that the number of iterates in a parallel loop is much larger than the number of processors P. If such is the case, it would be better to create only P threads and have each of them execute $(\frac{e_2-e_1}{e_3}+1)/P$ of the iterates in a serial loop. This procedure saves the overhead associated with spawning all the threads and increases the possibility of using fine-grain parallelism. ParC has a special construct that implements this optimization, called lparfor. Thus the use of lparfor in Fig. 2.7 is efficient and is equivalent to the parfor version of Fig. 2.8 where P is the number of processors.

Note, however, that the iterates must be independent for this procedure to work. lparfor explicitly implies that the iterations are independent. A correct program using parfor constructs may not be correct if lparfor is used instead. For example, the following parallel loop always halts, but not as an lparfor because the flag will never reset:

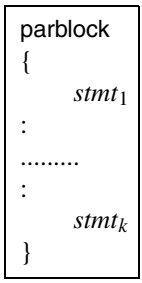


Fig. 2.9 The parblock construct

```
int x=1,z;
parfor(int i=0;i<n;i++)
   if(i == n-1){ z=f(); x=0;}
   else{
       while(x);
       A[i]=g(z);
   }
epar
```

Clearly one may always replace an lparfor with a parfor without affecting the correctness. The `lparfor` construct may be used to batch a number of small threads together in order to increase their granularity. However, such a procedure requires that these threads be independent.

The potential parallelism in recursion function calls and in a sequence of independent function calls $f(\ldots); g(\ldots)$ is handled by the parblock construct whose syntax is given in Fig. 2.9. For example it can be used to parallelize the structure of divide and conquer algorithms.[1] Note that the three parallel constructs are redundant: a parblock and lparfor may be implemented using a parfor, for example, but such implementations would be bulky and degrade the code quality significantly.

Since parallel constricts can be freely nested then there is a need to determine what happens when a thread spawns another thread. For example in Fig. 2.10 we should determine if the for-loop $for(j = i; j < i + 1000; j++)$ can continue execute its next iteration before all the threads spawned by the inner parfor (spawned in the current iteration of the $for(j = i; j < i + 1000; j++)$) terminated. In ParC the rule is that the thread that executes a parallel construct is suspended until all the constituent threads have terminated. Thus, the for-loop in Fig. 2.10 is blocked until the threads spawned in its last iteration terminate (yielding no more than 100×1000 threads running at any time of the execution).

Upon execution, nesting of parallel constructs thus creates a tree of threads where only the leaves are executing while internal nodes wait for their descendants.

We now present a set of simple programs in order to demonstrate how to use parallel constructs to create parallel programs. As an example of parfor usage, consider a program that determines if all the elements of an array are equal, Fig. 2.11.

[1] In the sequel we will also use the short notation of $PF\ i = 1 \ldots n\ [S_i]$ for the parfor construct and $PB\ [S_1, \ldots, S_k]$ for the parblock construct.

```
int array[100000][1000];
parfor(int i=0; i< 100000; i+=1000){
    int j;
    for(j=i; j<i+1000; j++)
        parfor(int k=0; k< 1000; k++) array[j][k] = 0;
}
```

Fig. 2.10 Nesting of parallel constructs

Fig. 2.11 Testing equality in parallel

```
int A[N];
int b=1;
parfor(int i=0; i<N; i++)
    if(A[i] != A[i+1]) b = 0;
```

Fig. 2.12 Testing equality in parallel using lparfor

```
int A[N];
int b=1;
lparfor(int i=0; i<N; i++)
    if(A[i] != A[i+1]) b = 0;
```

Fig. 2.13 Testing equality in parallel eliminating unnecessary comparisons

```
int A[N];
int b=1;
lparfor(int i=0; i<N; i++)
    if(A[i] != A[i+1] && b) b = 0;
```

The parallel program compares all elements in parallel. If one comparison fails, a flag (b) is set to zero. This program uses one parallel step instead of the N steps used by the sequential version. One can consider some issues of efficiency:

- Can we replace the parfor by a light lparfor? if so we improve the overhead for creating N threads (see Fig. 2.12).
- If we detected that two elements differ we need not complete the remaining iterations (see Fig. 2.13). Though (assuming that the b is tested first before $A[i] != A[i+1]$ is evaluated) we saved the comparison this version is still not efficient since it will execute N loop-iterations even if $b = 0$ is executed in an early stage (e.g., when $A[0] == A[1]$). Thus, the version in Fig. 2.14 is significantly better.

It is interesting to note that the effect of the version in Fig. 2.14 could have been obtained had we used break-statement as depicted in Fig. 2.15.

```
int A[N];
int b=1, np=N/P;
parfor(int ii=0; ii<N; ii+=np){
    int i;
    for(i=ii; (i<ii+np) && b; i++)
        if(A[i] != A[i+1] && b) b = 0;
    }
```

Fig. 2.14 Improving efficiency by simulating lparfor

```
int A[N];
int b=1;
lparfor(int i=0; i<N; i++){
    if(A[i] != A[i+1]) b = 0;
    if(b == 0) break;
}
```

Fig. 2.15 Using break-statement in a lparfor

```
int A[N];
int rsum(A,fr,to)
int A[],fr,to;
{
int a,b;
if(fr == to) return(A[fr]);
else
    parblock {
        a = rsum(A,fr,(to+fr)/2);
        :
        b = rsum(A,(to+fr)/2 +1,to);
    }
    return(a+b);
}
}
```

Fig. 2.16 Recursive parallel summing

As an example of parblock, consider a recursive routine that computes the sum of N elements of an array A, described in Fig. 2.16.

The parallel program computes the sums of each half of the array in parallel and then adds them together. Hence, it needs $2 \log N$ parallel steps compared to the $2N$ steps needed for the sequential version. The above program actually contains a bug and may not work correctly. Assume that $(to + fr)/2$ rounds the result up to $\lceil (to + fr)/2 \rceil$; then we have generated an infinite recursion, as depicted in Fig. 2.17.

Note that each call to *rsum*() is executed on a separate stack. Thus, when a thread terminates, its stack can be re-used when generating new threads. Clearly, had we used one stack to implement the calls of all the threads, we would suffer the following consequences:

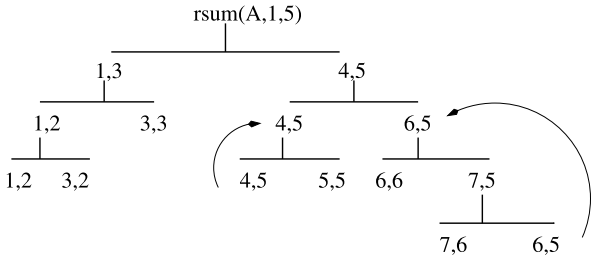


Fig. 2.17 Infinite recursion and incorrect sums can be obtained

```
int A[N];
int np=N/P,k,s=0,s_ar[P];
parfor(int l=0; l<P; l++)
{ int i;
    s_ar[l] = 0;
    for(i=l*np; i<(l+1)*np; i++)
        s_ar[l] += A[i];
}
for(k=0; k<P; k++) s += s_ar[k];
```

Fig. 2.18 Simple parallel programs for summing N numbers

- The stack would have been a sequential "bottle-neck" because each call would require us to "lock" the stack pointer, preventing other threads from performing function calls in parallel.
- If the function calls of the threads are mixed in one stack, whenever a function returns, it will leave a "hole" in the stack that will be hard to use.

The sum of N elements (see Fig. 2.18) can be computed more efficiently by a divide and conquer algorithm designed for P processors. The program divides A into P segments, computes the sum of every segment in parallel and stores it in a global array $s\_ar[l]$. The local sums are summed sequentially in P steps. Hence, the program needs $\frac{N}{P} + P$ parallel steps, compared to N needed by the sequential program. This version of *sum* is more efficient than *rsum* because it spawns fewer threads $P \ll N$ than in *rsum*.

This summing program contain some faults as follows:

- If N is not divided by P then some elements at the last part of the array will not be included in the summing.
- It is multiplying by *np* every iteration.

Figure 2.19 shows how to correct these problems.

2.3 Scoping Rules

In general scoping rules associate between definitions of variables in different scopes. Scoping rules determine which assignments can update these variables and

```
int A[N];
int np=N/P,k,s=0,s_ar[P];
parfor(int l=0; l<N; l+=np)
{ int i,z;
    z = l/np;
    s_ar[z] = 0;
    for(i=l; i<l+np; i++)
        s_ar[z] += A[i];
}
for(k=l; k<n; k++) s += A[k]; /* remaining un-summed elements (at most N/P-1 steps) */
for(k=0; k<P; k++) s += s_ar[k];
```

Fig. 2.19 Correct version of summing N numbers, in $O(N/P)$ steps

which expressions can use these variables. In general each function call creates a new activation record that holds its set of variables. Thus the scope of a given call is all the activation records of other function calls whose variables can be updated or used directly through their names by the current function call. This is depicted in Fig. 2.20 showing that in the case of a single threaded code there is only one stack and the scope of each accesses to a variable is uniquely defined by searching the last occurrence of a function's activation record on the stack. However, in a multi-threaded code there is a multiple set of stacks and as depicted in Fig. 2.20 it is not clear what is the scope of $f4$.

Note that *C* has only two levels of nesting variables: local to a procedure and global. *C*, however, also has a form of hierarchical scoping for variables because each block of code may contain a data declaration section. Thus, within a single procedure, there is hierarchical scoping. As the following example illustrates, in C the scoping rules are not fully enforced.

```
int g1;
for(int i=0;i<n;i++){
int g2;
    g2 = i+g1;
    if (A[i] < g2)
    { int g3;
      g3 = g3 + g2*i;
    }
}
printf(" is g3 accumulative? %d\n",g3);
}
```

In ParC, this hierarchical scoping is used to provide various degrees of sharing of variables among threads. In parallel languages, scoping is often augmented with explicit declarations that variables are either private or shared. We find that explicit declarations are unnecessary, as the scoping naturally leads to a rich selection of sharing patterns. Since ParC allows parallel constructs to be freely nested inside other parallel constructions it naturally creates nested patterns of shared/private variables.

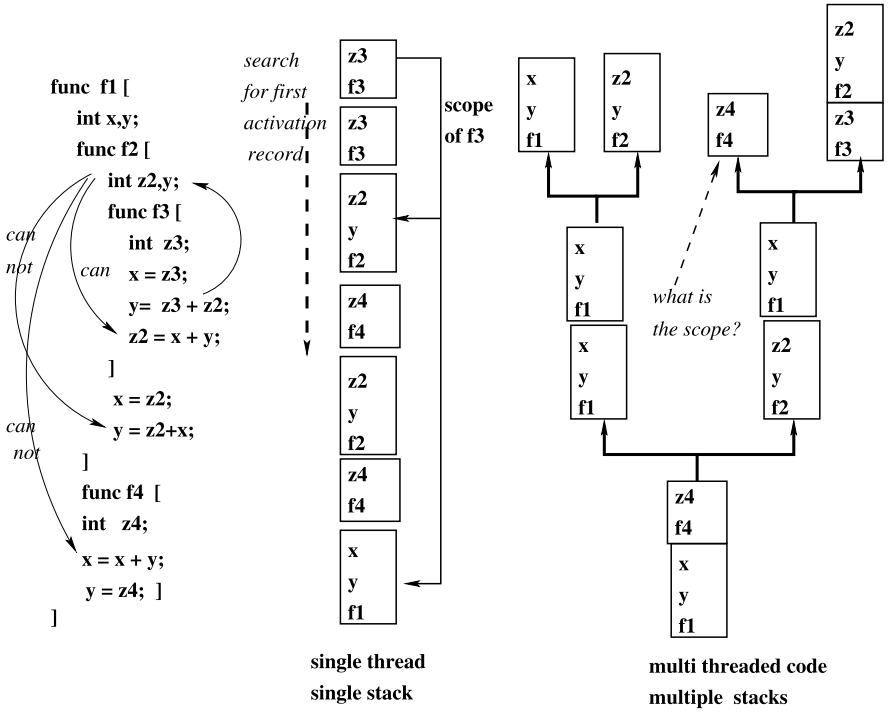


Fig. 2.20 Scoping rules in sequential environment versus scoping rules in multi-threaded code

As each iteration of a parfor/parblock creates a new thread we obtain multiple set of stacks for which the rule of:

```
Consider a reference to an ``external'' variable X made in f()
where X is defined in a different function F().
Search for the last occurrence of F() on the stack in order to
locate the right copy of X f() is referring to.
```

can not be applied.

ParC attempts to preserve the normal *C* language scoping rules. Thus, variables declared within a block (regardless of whether the block is parallel or sequential) can be accessed by any of its nested statements (parallel or sequential). This capability means that global variables are shared among all of the program threads; static variables and external variables are also considered global.

Local variables declared within a block are allocated to the thread's stack. The scoping rules, therefore, imply a logical "cactus stack" structure for nested parallel constructs, as each thread that is spawned adds a new stack that can be reused when the thread terminates. Note, however, that this is the case only for nested parallel constructs in the same procedure. Code in one procedure cannot access variables local to the procedure that called it unless they are passed as arguments. It cannot modify them unless they are passed by reference.

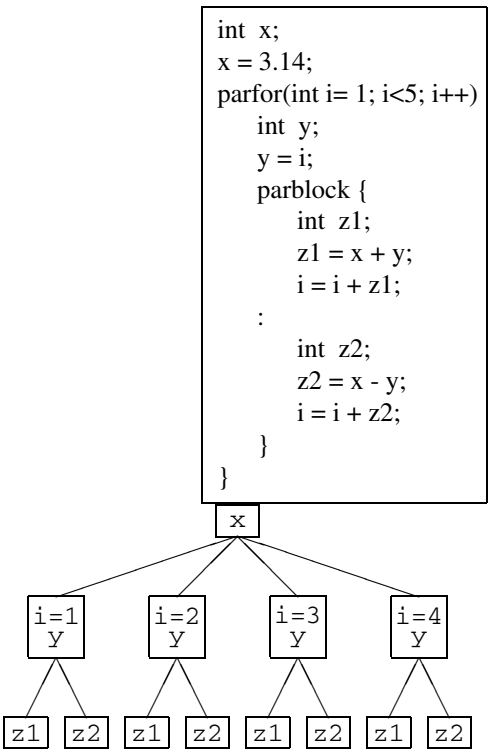


Fig. 2.21 Example to illustrate scoping rules

The array in the example of Fig. 2.4 is shared by all of the threads, while the index variable `i` is duplicated with a private copy in each thread. This is a special case of the different sharing patterns that are possible in *ParC*.

The scoping rules of *ParC* are the same as those of *C*: variables may be declared in any block, and their scope includes all blocks that are nested within the block in which they are declared. If some of these nested blocks are executed in parallel, the variable will be shared by all the decedent threads.

For example, consider the code segment in Fig. 2.21. The variable `x` is shared by all the threads. The `parfor` construct creates four threads. Each has a private copy of the index variable `i` that is initialized automatically by the system, and another private variable `y`. When the `parblock` is executed, each of these threads spawns two additional threads: one defines a private variable `z1` and the other a private variable `z2`. When one of these threads performs the instruction `z1 = x + y`, for example, it is adding the value of `x`, which is shared by everyone, to the value of `y`, which it shares only with its brother, and stores the result in a private variable that is inaccessible to other threads. The structure of the thread tree and the copies of the various variables are also shown in Fig. 2.21. Note that the assignments $i = i + z1, i = i + z2$ can affect each other only at the level of each parfor, but since i is duplicated there can be no effect of these assignments between the threads of the outer parfor.

The scoping rules described above are implemented by the ParC pre-compiler by changing the references to variables declared in surrounding blocks to indirect references through a pointer n(using $px = \&x;\ \ *px = y$ instead of $x = y$;). Thus, the run-time system does not have to:

- Chain stacks to each other explicitly to create a full-fledged cactus stack.
- Search along the set of stacks on the path from the current stack to the root stack to find the first occurrence of an activation record containing X for each external reference to X.
- Create special types of "shared addresses" to distinguish among the different copies of the local variables declared inside the parallel constructs. This is obtained through the use of indirect references via the $*px$ pointers.

Instead, direct pointers to the variables' location (typically on another thread's stack) are available. We call the resulting structure of stacks with mutual pointers into each other a *threaded stack*.

Figure 2.22 illustrates the implementation of the scoping rules with threaded stacks. The two threads in the parblock share the variables `i,x,A[]` (but use only `x,A`) declared before the parblock in the external parfor. Each parfor iteration or parblock statement is transformed by the ParC compiler into a function ($f1$, $f2$, $f3()$ in the figure) that receives a pointer to `i` as its argument. The function names and the arguments are passed as parameters to the `spawnPB`, `spawnPF` functions. These spawn-functions are part of the thread system library. When it is called, it creates new threads on different processors/cores. As depicted in Fig. 2.22, each thread is complete with its own call stack. An activation record for the thread's function is constructed on the thread's stack, and the appropriate arguments are copied to it. The thread can then commence and behaves as if the function were called on a remote processor/core. Thus due to the execution of the program in Fig. 2.22 the *spawnPF* will generate four threads each with its own stack ($f3(0)$, $f3(1)$, $f3(2)$, $f3(3)$ in the figure). Each of these four threads will spawn two new threads of the parblock yielding $1 + 4 + 8$ threads and stacks.

The issue of managing the cactus stack efficiently and the overhead involved with it is an important consideration when designing a multi-threaded system. The cactus stack is a direct consequence of the ability to nest parallel constructs and to mix them with recursive calls. These abilities form the main core of a free non-restricted parallel programming style. An alternative approach (for obtaining more efficient handling of stacks) is to sacrifice nesting + recursion and restrict the programmer to using a fixed set of P threads for the whole execution. This programming style is called Single Program Multiple Data (SPMD) and is equivalent to the following restrictions in ParC:

1. Use only *parfor* $(inti = 0; i < P; i++)\{Seq - C - CODE\};$.
2. Nesting of parallel constructs is not allowed.
3. *parfor* ()s can be used only in *main*().
4. The execution of parfor () should be unconditional.
5. Threads can communicate only through a global set of common shared variables.

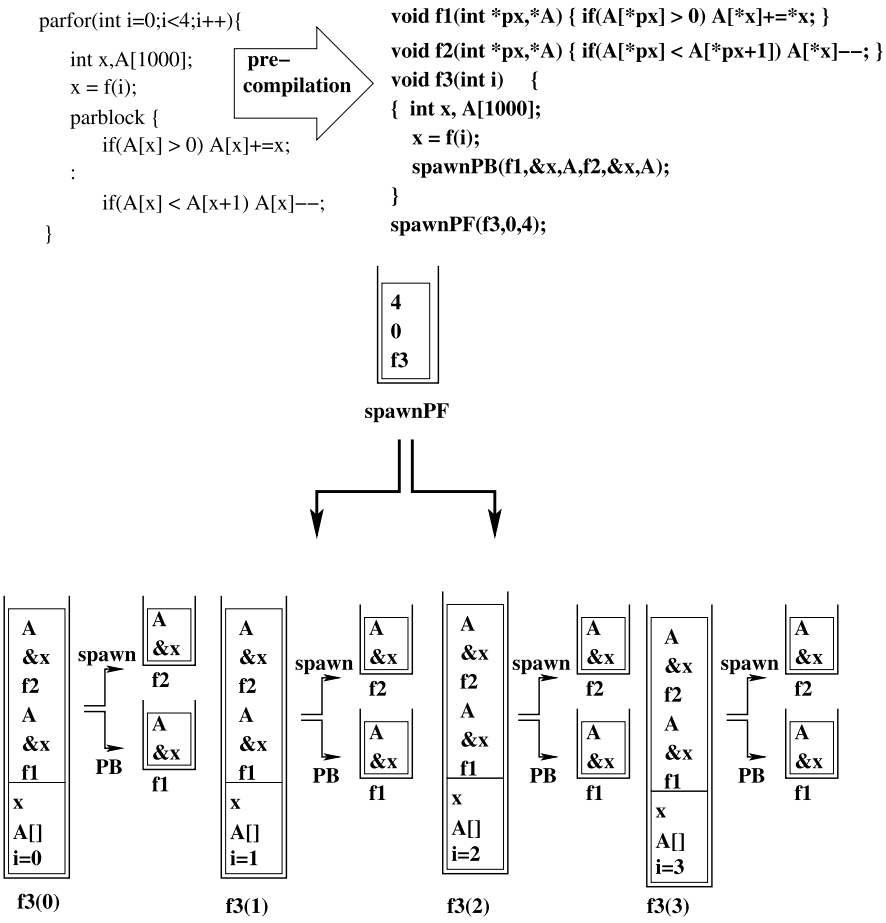


Fig. 2.22 Compilation of a ParC program and the resulting cactus stack generated by its execution

Thus, the recursive summing cannot be programmed in SPMD style. In SPMD style there is no need to support the cactus stack. Each iteration of the chain of parfor ()s composing the SPMD program is executed by one of the threads of the fixed P threads allocated at the beginning of the execution using its local stack. Figure 2.23 illustrates this fact showing that a set of three separate stacks can be used to implement a parallel program that agrees with the requirements of SPMD. In fact, as depicted in Fig. 2.23 we can execute an SPMD program using a fixed set of threads with only shared memory to hold global shared variables.

2.3.1 Using Static Variables

In *C*, static variables are declared inside functions but are shared or common to all invocations (calls) made to this function. The implementation is, therefore, to

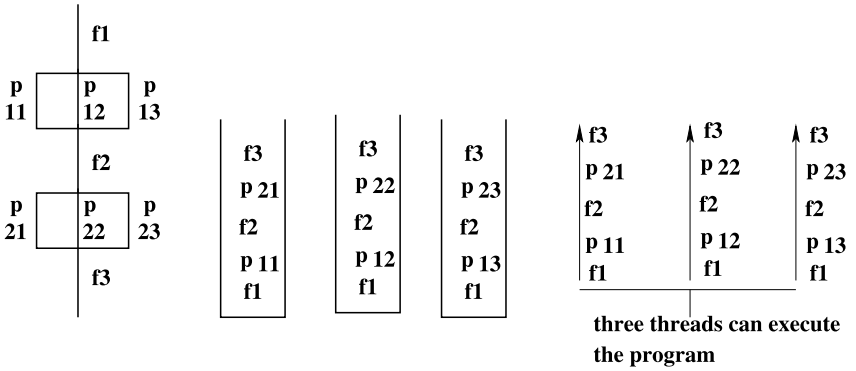


Fig. 2.23 SPMD allow to use separate flat set of stacks

```
parfor ( j; 0; N−1; 1 ) {
    parfor ( i; 0; N−1; 1 ) {
        static int s=0;          /* shared by all N threads * /
        int t;                   /* private to each thread * /

        s++;                     /* atomic incremental of 's' * /
        t++;                     /* private variables need not be protected * /
    }
}
printf("s will be N*N%d",s);
```

Fig. 2.24 A variable declared as static within the construct will actually be shared by all the threads of the construct!

allocate storage for such variables from the global heap space, rather than on the stack. In *ParC*, static variables declared in a parallel block become shared by all the threads that activate a function with that static variable (see Fig. 2.24).

2.3.2 Bypassing the Scoping Rules

The scoping rules of *ParC* make local variables declared in different threads separate entities. The goal is to let each thread use its own local copy safely without fear of being updated by another thread. However, due to the ability of C to pass the address of a variable through a global variable, it is possible to bypass the locality, allowing one thread to change the value of a local variable of another thread which is normally not visible to it. In regular C code one can use this ability to produce harmful effects such as accessing a local variable of a function after it has returned:

Fig. 2.25 Bypassing locality using a shared variable

```
int  *p;
     parfor( int i=1; i<4; i++)
     {
          int  Y;
          Y = i;
          P = &Y;
          *P = i;
          printf("<%d,%d>",i,Y);
     }
```

```
int *g,A[100];
int f(int z){
   int x,y;
   x= A[z];
   y= A[x];
   g = &y;
   return(x+y);
}

main(){
   *g = f(5);
}
```

In *ParC* this ability allows us to bypass the scoping rules, an ability that is potentially harmful, but can also be useful. Figure 2.25 demonstrates this ability by showing three threads that can potentially affect the local variables ($Y_{1,2,3}$) of one other. Let Y_i denote the local copy of the ith thread. One could expect that the program will print $\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle$ in different orders. However, other results are possible because one thread can change the value of Y of another thread if P has its address. For example $\langle 1,1\rangle\langle 2,3\rangle\langle 3,3\rangle$ will be printed by the sequence:

$$Y_1 = 1;\ Y_2 = 2;\ Y_3 = 3;\ P = \&Y_1;\ P = \&Y_3;$$
$$P = \&Y_2;\ Y_1 = *P = 1;\ Y_2 = *P = 2;\ Y_3 = *P = 3; \quad \langle 1,1\rangle\langle 2,3\rangle\langle 3,3\rangle$$

Or $\langle 2,1\rangle\langle 1,3\rangle\langle 3,3\rangle$ will be printed by the sequence:

$$Y_1 = 1;\ Y_2 = 2;\ Y_3 = 3;\ P = \&Y_3;\ Y_3 = *P = 3;\ P = \&Y_1;\ Y_1 = *P = 3;$$
$$P = \&Y_2;\ Y_2 = *P = 1; \quad \langle 2,1\rangle \qquad Y_2 = *P = 2; \quad \langle 1,3\rangle\langle 3,3\rangle$$

2.4 Effect of Execution Order and Atomicity

Another factor that should be considered is the fact that ParC assignments are not **atomic**. An operation is atomic if it is executed in one step by the hardware, and

its result cannot be affected by the parallel execution of another operation. In ParC, the assignment $X = X + 1$ should be divided into atomic *load* and *stores* operations such that interleavings with other operations can take place. Hence, an instruction is atomic if it cannot be interleaved with any other operation. In order to compute all possible results of a ParC program, the user has to divide every expression and assignment into a sequence of load and stores, and then find all of the possible interleavings or execution orders.

The next program attempts to compute the sum of N elements in one step. If $X = X + 1$ had been atomic, in all execution orders X would have contained the sum of all of the elements in A. However, $X = X + 1$ is not atomic. Its atomic version is presented on the right side of the figure below. Clearly, there are execution orders where X will contain the value of the last store ($x = t$;). In other words, instead of $\sum A[I]$, X will contain the value of some $A[I]$.

```
int  A[N],X=0;
parfor( int i=1;i<N;i++){
     X = X + A[I];
}
```

False summing program.

```
int  X=0;
parfor( int i=1;i<N;i++){
     int t,*a,ai;
     t = X; /*load x*/
     a = A; /* load A[I] */
     a = a +I;
     ai = *a;
     t = t + ai;
     X = t; /*store */
}
```

An atomic version demonstrating harmful execution orders.

Clearly, interleaving instructions from different threads (execution order) may have an effect on program termination, in addition to the obvious effect on access to shared variables.

2.5 Parallel Algorithms for Finding the Minimum

In this section we study several basic parallel algorithms demonstrating some basic parallel programming techniques. These techniques are elementary but general enough so that they can be viewed as parallel programming techniques rather than specific algorithmic solutions to a single problem. Programming, testing and comparing different variants of the same problem may lead to a deeper understanding of the programming gap between algorithms and their implementation as parallel programs. The underlying algorithms solve the problem of finding the minimum of N elements stored in the array. Most of these algorithms are designed for a given

```
input A[N];
temporary partial_res[P];
parfor(int l=0; l<N; l+=t)
{ int i,z;
    z = (N/P);
    partial_res[l/z] = 0;
    for(i=l; i<l+z; i++)
        partial_res[l/z] op= solve(A[i]);
}
for(k=0; k<P; k++) combine(partial_res[k]);
```

Fig. 2.26 Basic scheme of divide and conquer using $k = P$ parts

number of processors. The algorithms vary in the number of processors needed. For example, when P (number of processors) is N^2, the minimum of N elements can be found in few parallel steps while for $P \leq N$ it takes about $\log N$ steps. When these algorithms are implemented in *ParC*, processors become threads. Hence, the number of processors used by the algorithm can affect the expected performance because of the overhead associated with the threads' creation.

2.5.1 Divide and Conquer Min

This solution is based on the summing algorithms described earlier (Fig. 2.16). A summing algorithm can be converted into an algorithm for finding the minimum by replacing the $+(x, y)$ operation with a $min(X, Y)$ operation. This procedure yields algorithms that use up to N threads and finds the minimum in $O(\log N)$ steps.

Basically, in this sort of solution the problem's input is split into $k > 1$ parts each is solved in parallel and independently of the rest of the parts. Finally a combine (conquer stage) finds the solution of the problem by processing all the results of k parts (sequentially or in parallel). Figure 2.26 illustrates the general scheme for divide and conquer when $k = P$.

2.5.2 Crash Min

The assumption used in this simple algorithm is that the number of available processors is N^2, where N is the problem size. In this case the power of common read and write to the same cell can be used to find the minimum in one step. The crush technique relies on the ability to use one shared memory cell to compute the boolean functions (e.g., AND) of n bits in one step as follows:

```
int b,A[N];

b = 1;
parfor(i=0;i<N;i++){
   if (A[i] == 0) b = 0;
}
printf(" the AND of A[1] to A[N] is %d",b);
```

Parallel AND of N bits

The crush method for finding the minimum of $A[N]$ can be described by picturing an $N \times N$ array $C[N][N]$ used to store the results of the comparisons between all elements of $A[N]$. For each i, j a TRUE is placed in $C[i][j]$ if $A[i] \leq A[j]$. If such is not the case, a FALSE is placed there instead. One or more FALSEs in the ith row of $C[][]$ indicates that the ith element is not the minimum. A parallel AND can be used to identify rows in $C[][]$ containing only TRUEs, a procedure that can be done in parallel by N^2 threads. The minimum is obtained by executing $min = A[i]$ for all rows in $C[][]$ with TRUE elements only. In fact, we need to check only the cells (in the imaginary array) that are above the main diagonal, so we need only $N \cdot (N-1)/2$ threads.

```
#define F 0
#define T 1
crush( ar, size )
int *ar,size;
{ int result[MAX],min;
   parfor(int i=0; i<size; i++)
      { result[i] = T; }
   parfor(int i=0; i<size-1; i++)
   {
      parfor(int j=0; j<size; j++)
      {
         if( ar[i] < ar[j] ) result[j] = F;
      }
   }
   parfor(int i=0; i<size; i++)
   {
      if ( result[i] == T ) min = ar[i];
   }
   return( min );
}
```

The crush method of finding the minimum in a constant number of steps

2.5.3 Random Min

The random algorithm uses N processors and advances in stages. In each stage there is a candidate for the minimum denoted by *min*. All threads whose element is greater than or equal to *min* are terminated because their value $A[i]$ is not the minimum. A set of $\sqrt{N}$ elements is chosen from the remaining candidates. Since there are N processors (or threads), the crash method can be used to find the minimum of this set. This minimum becomes the new candidate and the process repeats itself until all of the elements are greater than or equal to *min*. A simple probabilistic analysis shows that 6 steps are quite probably all that are needed.

```
#define F 0
#define T 1
random_min( ar , size )
int *ar,size;
{ int min,tag,sqr[ SQRMAX ];
   parfor(int i=0; i<SQRMAX; i++)
   { sqr[ i ] = BIGNUM; }
   for( min = BIG , tag == T; tag = T;)
   {
      tag = F;
      parfor(int i=0; i<size; i++)
      {
         if( ar[i] < min )
         {
            tag = T;
            sqr[ rand( SQRMAX ) ] = ar[ i ];
         }
      }
      min = crush( sqr , SQRMAX );
   }
}
```

Random method of finding the minimum in a constant number of steps

A simple calculation can be used to illustrate why this algorithm takes no more than a constant number of iterations.

- Let n be the size of ar[].
- Since we are taking a random sample of $\sqrt{n}$ elements of the array, we may assume that the sample comes from a sorted array where the smallest element is ar[0].

- The probability that an element chosen at random from $ar[]$ will not "fall" into the first segment of the smallest $\sqrt{n}$ elements is $1 - \frac{1}{\sqrt{n}}$. We use a sample of size $\sqrt{n}$. Thus, the probability that none of these elements will fall into the smallest $\sqrt{n}$ element is $(1 - \frac{1}{\sqrt{n}})^{\sqrt{n}} = \frac{1}{e}$.
- By repeating this process a constant number of times, we can reduce the above probability to any desired constant fraction, e.g., $\frac{1}{e^{100}}$.
- Clearly, once the sample contains an element in the smallest $\sqrt{n}$ elements, the number of remaining elements in $ar[]$ after the comparison with *min* is less than $\sqrt{n}$. Consequently, the next application of *crash*() will find the correct minimum.

Note that we used a "trick" to select a random sampling of the remaining elements in $ar[]$. Hypothetically, we should have "packed" the remaining elements and generated a random permutation of them, which would have been a complex task. However, we used the fact that the results of parallel writes to $sqr[]$ are arbitrary to approximate a random sampling. In other words, when $k > \sqrt{n}$ elements are written in parallel to random locations in $sqr[]$, we assume that there is an equal probability that each of these k elements will not be "erased" by another write to the same location. This technique of using random mapping to "pack" $k \leq N$ elements from a larger array $A[N]$ into a smaller array $B[k]$ is based on the well known claim:

Claim 2.1 *Consider a process where L balls are thrown at random into k jars. If $L \geq k \cdot 2 \cdot \log k$, the probability that there will be an empty jar is less than $\frac{1}{k}$.*

The probability that a ball will hit a given jar is $(1 - 1/k)$. The probability that all $k \log k$ balls will miss that jar is $(1 - 1/k)^{k \log k}$. Given that $(1 - 1/k)^k = \frac{1}{e}$, this probability is less than or equal to $\frac{1}{e^{\log k}}$. Thus, for $L \geq k \cdot 2 \cdot \log k$, the above claim follows.

2.5.4 Divide and Crush

Assume that the number of processors P is $2 \cdot N$, and N is a power of 2. In each step, we divide the input into groups such there are enough processors for each group to execute the *crash*(). For the next step, we keep only the minimum elements of each group. In each step we have fewer minimum (candidates) and, therefore, more processors per group. Hence, the size of every group is the square size of the groups of the last step. Thus, the algorithm should terminate in $\log \log N$ steps. The last iteration takes place when the number of items left is smaller than $\sqrt{N}$. In Table 2.1 we show the relationship between group size, number of iterations and number of processors needed per group.

Table 2.1 Relationships between sizes in the divide and crush algorithm

Step	Items	Group size	# of groups	Processors per group
0	N	$2 = 2^{2^0}$	$N/2$	$2N/(N/2) = 4$
1	$N/2$	$4 = 2^{2^1}$	$N/(2 \times 4)$	$2N/(N/8) = 16$
2	$N/8$	$16 = 2^{2^2}$	$N/(8 \times 16)$	$2N/(N/(8 \times 16)) = 256$
3	$N/128$	$256 = 2^{2^3}$	$N/(128 \times 256)$	$2N/(N/(128 \times 256)) = 256 \times 256$
4	…			
5	…			

```
div_crush( ar , size )
int *ar,size;
{
   int b[ N/2 ];
   int gs; /* group size */
   int nbg; /* number of groups */
   int min; /* minimum value */
   gs = 2; nbg = size/2;
   while( size >= gs )
   {
      parfor(int i=0;i<nbg;i++)
      { /* find minimum of each group */
         b[i] = crash( &ar[ i * gs], gs);
      }
      parfor(int i=0;i<nbg;i++)
      { ar[i] = b[i]; }
      size = nbg;
      gs = gs * gs ;
      nbg = size/gs;
   }
   if( size == 1 ) return( ar[0] );
   min = crush( ar, size );
}
```

Divide and crush method for finding the minimum in $\log\log N$ steps

The following argument by Valiant's demonstrates that this result is optimal. Valiant considered a "stronger model" for algorithms that computes the minimum of n elements using P processors. In this model, at every stage the algorithm can perform only P comparisons and then, **based on the results of these comparisons alone**, remove those elements that are clearly not the minimum. This is a very re-

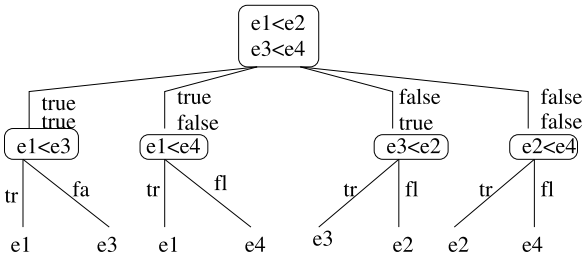


Fig. 2.27 A decision tree for finding the minimum of four elements with $P = 2$

stricted model, a fact that becomes evident when we model it using a game between two players: the algorithm and the adversary. The algorithm player can ask the adversary only about comparisons between two elements. The adversary, on the other hand, does not even have to provide values for the n underlying elements. All he must do is give consistent answers. The only question is how long the adversary can delay the algorithm before the minimal element is located. The game's rules preclude the algorithm from using any other type of information (e.g., the value of some elements or their total sum). However, this is a natural restriction and a common way of obtaining lower bounds in computer science.

Another way of viewing such a restricted family of algorithms is as a decision tree wherein (see Fig. 2.27):

- Each node (except the leaves) represents a query of up to P comparisons involving up to $2P$ elements out of n initial elements.
- there are up to 2^P outgoing edges from every internal node marked by all possible outcomes of these comparisons.
- Each leaf is marked by a decision as to which element is the minimum.
- Such a decision tree is "legal" if one cannot find n elements $e_1, \ldots, e_n$ such that the decision in some leaf is "wrong."

We can obtain the lower bound by showing that the height of any decision tree is greater than $\log \log n$ for any decision tree that finds the minimum of n elements. Note that after each query the algorithm "knows" the transitive cluster of all the comparisons that were made up to this point. Thus, we can conclude that after each query the algorithm can filter out the elements that are clearly greater than some other elements. After each comparison, the algorithm can focus only on those elements that have not been compared to any other element or did not "lose" in any comparison.

Every comparison between two elements u, v can be represented by a "directed edge" $\langle u, v \rangle$ (the direction depends on the result of the comparison). Thus, at any stage of an algorithm A we can represent the computation by a comparison graph G_A. For example, applying *crash*() on a group of elements yields a graph with an edge between every two elements ("clique") because each element is compared with all of the other elements. As explained before, after the edges have been added, we perform a transitive closure of the graph's end to remove all the elements that did

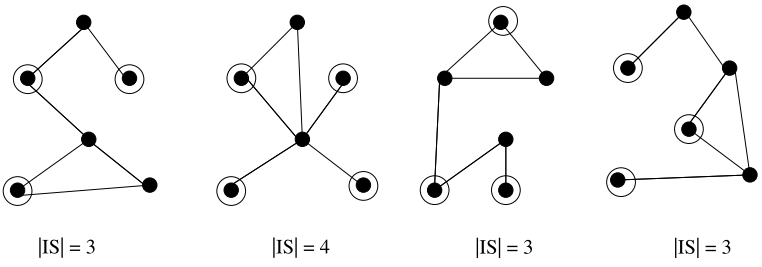


Fig. 2.28 Illustration of Toran's theorem

not win all of their comparisons (i.e., if $\langle u, v\rangle$ is an edge and $\langle v, w\rangle$ is an edge, then $\langle u, w\rangle$ is an edge). A given algorithm A can terminate when G_A does not contain two elements that have not been compared to each other (either directly or indirectly). This follows from the fact that if there are two uncompared elements (u, v) in G_A, then the minimum can be either u or v. Thus, we could have assigned values to the elements in such a way that the values would be consistent with all of the comparisons in G_A yet $u < v$ or alternatively chosen another set of values (equally consistent with G_A) with $v < u$.

Let an independent set *IS* in G_A be a set of nodes that are not connected to each other in G_A (after the transitive cluster, but before the filtering). An *IS* is maximal if we cannot add more nodes to it without violating the previous condition. It follows that for every algorithm, the size of the remaining elements (the remaining candidates for the minimum) is at least the size of the maximal *IS* in G_A. For example, the size of the maximal IS (MIS) of a directed chain with n nodes is 1 because after the transitive closure we have edges from the first node in the chain to all of the others.

The size of any maximal independent set in a directed graph G can be bounded from below by using a well known theorem by Toran:

Theorem 2.1 *The size of any maximal IS in a directed graph with n nodes and* $m \leq n$ *edges is greater than or equal to* $\frac{n^2}{2 \cdot m}$.

Consider, for example, the set of graphs in Fig. 2.28 showing that for $n = 6$ and $m = 6$ there is always an independent set with at least $\frac{6^2}{2 \cdot 6} = 3$ nodes.

Applying Toran's theorem to the minimum algorithm with $P = n$ processors implies that after the first query of up to P comparisons the size of the *IS* (remaining elements) in any G_A is greater than or equal to

$$a_0 = \frac{n^2}{2 \cdot P}$$

Thus, after the next query the size of the remaining elements is greater than or equal

to

$$a_1 = \frac{{a_0}^2}{2 \cdot P} = \frac{n^4}{4 \cdot P^2 \cdot 2 \cdot P} = 2 \cdot P \left(\frac{n}{2 \cdot P} \right)^4$$

and in general

$$a_t = 2 \cdot P \left(\frac{n}{2 \cdot P} \right)^{2^t}$$

The algorithm cannot terminate before $a_t = 1$. Thus, for $P = n$

$$1 = 2 \cdot n \left(\frac{n}{2 \cdot n} \right)^{2^t}, \qquad \log 1 = \log^2 n + 2^t \cdot \log \frac{1}{2}, \quad 2^t = \log^2 n, \; t = \log \log n$$

2.5.5 Asynchronous Min

The underlying machine is an asynchronous one, so the scheduling in the execution of the previous algorithms can be different from the expected PRAM synchronous one. A simple method that is oblivious to the execution order is the *async_min*. In this method, a tentative minimum is held as a global variable. Each element $A[i]$ compares itself repeatedly to that global minimum. If it is bigger, then the thread terminates itself. Otherwise, it tries to update the tentative minimum to its own value. The algorithm terminates when there is no element bigger than the tentative minimum. A thread that has been suspended from the ready queue to the suspend queue (see Fig. 1.13) may benefit from the fact that a smaller element has updated the tentative minimum in that it may be spared some of its iterations. The algorithm is expected to terminate in $\log N$ steps because the probability that an element that has updated the tentative minimum is bigger than half of the remaining elements is high.

```
sync_min( ar , size )
int *ar,size;
{
int b=1,tent_min = MAX_VAL;
   while(b == 1) {
      parfor(int i=0;i<N;i++)
      {
         while ( A[i] < tent_min ) tent_min = A[i];
      }
      b = 0;
      parfor(int i=0;i<N;i++)
      {
         if ( A[i] < tent_min ) b = 1;
      }
   } }
```

Asynchronous minimum algorithm with expected time $\log N$.

As will be explained later on, if we join the two parallel-for statements, the program will not be correct and may compute the wrong result.

2.6 Recursion and Parallel Loops: The Parallel Prefix Example

The parfor and parblock constructs may be nested in arbitrary ways, creating a tree of threads where all the leaves are executing in parallel. This section illustrates how to directly compute $T_p(A(n))$ in a case where recursion and parallel for-loops are combined, where T_p is the number of parallel steps needed to solve a problem A with input size n by p processors. In this section T_p is a formula (probably recursive) that informally describes the number of synchronous parallel steps needed to compute the problem. Note that we cannot simply use the previous execution model and generate the set of all possible time diagrams of a ParC program, as this is simply not a "constructive" method to compute the time. The previous execution models should be regarded as proofs that execution time can be attributed to ParC programs. In this section we briefly consider the problem of obtaining an approximate formula to the execution time assuming that:

- There are no while-loops.
- The number of iterations each loop is executing is a known function of the input size.
- The number of threads that is spawned in every construct is also a known function of the input size.
- The input values do not affect the number of iterations in loops, thread generation and so forth.
- Synchronous execution of the program, step-by-step such that in each step the p processors are simulating a desired number of $P > p$ processors needed to execute all the instructions of the next step.

The program in Fig. 2.29 illustrates a combination of parfor and parblock inside parallel recursion. The program computes the partial sums of an array such that each cell is the sum of all cells that precede it (including itself). In other words, $A[i] = \sum_{j=0}^{j \le i} A[j]$.

In order to estimate $T_p(prefix(n))$ one can use the following recursive equation:

$$T_p(n) = T_p\left(\frac{n}{2}\right) + \frac{n}{2p} + 1$$

where $T_p(\frac{n}{2})$ is the time needed to compute in parallel the partial sums of every half; $\frac{n}{2p}$ is the time needed to add the sum of all of the elements of the left half to all of the elements of the right half, using p processors; and $+1$ indicates the local computations such as $if(beg \ge end)\ldots$; and the computation of m. The solution for this equation is:

$$T_p(1) = 1 \qquad T_p(n) = \frac{n}{2p} + 1 + T_p\left(\frac{n}{2}\right) = \frac{n}{2p} + 1 + \frac{n}{4p} + 1 + \cdots = \frac{n}{p} + \log n$$

Fig. 2.29 Implementation of parallel prefix computation using divide and conquer

```
prefix( arr, beg, end )
int arr[], beg, end ;
{
      int m;
      if (beg ≥ end) return;
      m = beg + (end - beg + 1)/2 -1;
      parblock
            { prefix( arr, beg, m ); }
            :
            { prefix( arr, m+1, end ); }

      parfor(int i= m+1;i <= end; i++)
      {
            arr[i] = arr[i] + arr[m];
      }

}
```

Note that this execution time is close to optimal in the sense that $\frac{n}{p}$ is a lower bound because each input must be read. The second term is also a lower bound because, as we proved in Sect. 1.6.3, using only summing operations, the value of the last element $A[n]$ cannot be computed in less than $\log n$ sequential sums. Thus, $T_p(n) = lowerbound_1 + lowerbound_2$, which is the optimal possible, as it is at most twice $max(lowerbound_1, lowerbound_2)$. We may even argue that an execution time which is the sum of lower bounds implies that we have a very efficient parallel program (even in a simplified model that does not take into account too many practical factors such as counting cache misses). Moreover, the additive factor $\log n$ is also an optimal factor because when n is increased or p is increased (but not both), $\frac{n}{p}$ dominates the $\log(n)$ factor.

Though a desirable estimation, the above computation of T_p is erroneous. This equation ignores the fact that the p processors should be divided between the recursive parallel computations used by the parallel prefix program. Assuming an equal division of the processors:

$$T_p(n) = \begin{cases} 1 + 2T_1(\frac{n}{2}) + \frac{n}{2} & p = 1 \\ 1 + T_{\frac{p}{2}}(\frac{n}{2}) + \frac{n}{2p} & p > 1 \end{cases}$$

This equation is solved by continuously expanding the second case $p > 1$ until after $\log p$ iterations, $p = 1$ and we are left with $T_1(\frac{n}{p})$:

$$T_1(n) \approx 2 \cdot T_1\left(\frac{n}{2}\right) + \frac{n}{2} = 4 \cdot T_1\left(\frac{n}{4}\right) + \frac{n}{2} + \frac{n}{2} = \cdots = \frac{n \log n}{2}$$

```
prefix( arr,beg, end )
int arr[n], beg,end ;
{
      int m,tmp[n];
      if (beg ≥ end) return;
      m = (end + 1)/2 -1;
      parfor(int i= 0; i<=end; i+=2)
      {
            tmp[i/2] = arr[i] + arr[i+1];
      }
      prefix( tmp, m );
      parfor(int i= 0; i<=end; i+=2)
      {
            arr[i] = tmp[i/2]; /* even locations are ready as is*/
            arr[i+1] = tmp[i/2] + arr[i+1]; /* odd locations must be updated*/
      }
}
```

Fig. 2.30 A better algorithm for computing prefix sums with an additive factor of $\log n$

$$T_p(n) = 1 + \frac{n}{2p} + 1 + \frac{\frac{n}{2}}{2\frac{p}{2}} + T_{\frac{p}{4}}\left(\frac{n}{4}\right)$$
$$= 1 + \frac{n}{2p} + 1 + \frac{n}{2p} + 1 + \frac{\frac{n}{4}}{2\frac{p}{4}} + T_{\frac{p}{8}}\left(\frac{n}{8}\right)$$

after neglecting minor factors such as using $\frac{n}{2p}$ instead of $(1 + \frac{n}{2p})$ (assuming $\frac{n}{2p} > 1$, we get

$$\log p\left(1 + \frac{n}{2p}\right) + T_1\left(\frac{n}{p}\right) = \log p\left(\frac{n}{2p}\right) + \frac{n}{p}(\log n - \log p) \approx \frac{n}{p}\log n$$

This last formula has a multiplicative factor of $\log n$ and thus is less desirable from a practical point of view than a formula with an additive factor of $\log n$. Such a formula can be obtained if we choose a different algorithm. The proposed "odd-even-prefix" algorithm (see Fig. 2.30) first sums each two consecutive elements (odd and even)in parallel. The result (half the size of the original array) is stored in a temporary array. Next, the algorithm recursively computes the prefix sums of the temporary array. Finally, the prefix sums of the original array can be computed in parallel by subtracting the original even element from the appropriate element of the temporary array.

Note that the parallel prefix can be computed in $T_p(n) = \frac{2n}{p} + p$ using the program in Fig. 2.31. The program divides the array into p chunks of $\frac{n}{p}$ elements and

Fig. 2.31 Efficient version of the parallel prefix

```
int A[N];
int k,ar[P];
      parfor(int l=0; l<N-1; l+=N/P)
      { int i;
            ar[l] = 0;
            for(i=l; i<l+N/P-1; i++)
                  ar[l] += A[I];
      }
      for(k=1; k<P; k++) ar[k] += ar[k-1];
      parfor(int l=1; l<P; 1++)
      { int i;
            A[l*(N/P)] += ar[l];
            for(i=l*(N/P)+1; i<(l+1)N/P-1; i++)
                  A[i] += A[i-1];
      }
```

computes the partial sums of every chunk in parallel. The parallel prefix of the partial sums is computed sequentially in $ar[i]$. These partial sums are added to the elements of every chunk while computing the partial sums of every chunk.

The parallel prefix is a basic operation in many parallel computations. Consider, for example, packing the non-zero elements of an array $A[N]$ into another array $C[]$. Intuitively, the position of the element $A[i] > 0$ in the packed array $C[]$ is exactly equal to the number of non-zero elements of $A[0, \ldots, i]$ or the prefix sum of $B[i]$ where $B[i] = 1$ if $A[i] > 0$, and zero otherwise (see Fig. 2.32). Many parallel programming languages such as OpenMP and MPI contain special constructs that implement prefix operations called reductions as primitive operations in the language. Typically, the operation for reductions can be set by the user to any associative accumulative operation such as multiplication or the max-operation. Note that we used the associativity of the '$+$' operation when we performed parallel prefix operations on different parts of $A[]$.

Clearly, the effect of a limited number of processors is significant, and one should carefully divide the processors among the different parallel computations spawned in a program. Note that such a direct computation of time can only be an approximation because the assignment of threads or computations to processors is not known and is done dynamically by the underlying operating system. Hence, the execution time of a program cannot be calculated, but some estimations can be computed, taking into account the effect of scheduling and other significant factors. For example, consider the program in Fig. 2.33, where a depth-first scheduling will produce 1000 threads, while a breadth-first scheduling may produce 2^{1000} threads. Clearly, $T_p(f)$ is affected by the scheduling policy of the underlying operating system.

In spite of the dependency of T_p on the scheduling, one could define the T_p of a program, assuming a "natural" breadth-first scheduling where processors are divided equally among the different computations:

```
int A[N],B[N],C[N];
int k,ar[P];
      lparfor(int i=0; i<N; i++)
      { if(A[i] > 0) B[i]=1; else B[i]=0; }
      prefix(B,0,N-1);
      lparfor(int i=0; i<N; i++) {
            if(i==0&& B[0] > 0]) C[0]=A[0];
            else
            if(B[i] > B[i-1]) C[B[i]-1]=A[i];
      }
```

Fig. 2.32 Packing the non-zero elements of an array

```
int  x=1;
f(x);

f(y)
int  y;
{
      x = y;
      if (x<1000)
      parblock
      {
            f(y+1);
            :
            f(y+1);
      }
}
```

Fig. 2.33 The effect of scheduling on the execution time

$$T_p(atomic - statement) = 1$$

$$T_p(LOOP\ i = 1 \ldots k\ S_i) = \sum_{i=1}^{k} T_p(S_i)$$

$$T_p(\{S_1, \ldots, S_k\} = \sum_{i=1}^{k} T_p(S_i)$$

$$T_p\big(if\,(cond) S_1\ else\ S_2\big) = \max\left(T_p(S_1), T_p(S_2) + 1\right.$$

$$T_p\big(PF\ i = 1 \ldots k[S_i]\big) = \begin{cases} \max_{i=1}^{k} T_{\frac{k}{p}}(S_i) & p \geq k \\ k \cdot \max_{i=1}^{k} T_{\frac{k}{p}}(S_i) & p < k \end{cases}$$

$$T_p\big(PB[S_1,\ldots,S_k]\big) = \begin{cases} \max_{i=1}^{k} T_{\frac{k}{p}}(S_i) & p \geq k \\ k \cdot \max_{i=1}^{k} T_{\frac{k}{p}}(S_i) & p < k \end{cases}$$

$$T_p(function - call) = T_p(body\ of\ the\ function + parameters) + 1$$

Note that this definition determines that processors are equally distributed among all the parallel computations of the parfor or parblock statements. This determination is made regardless of the **real** number of processors that a computation requires. Hence, if two computations are spawned where 4 processors are available, then each computation will receive 2 processors. However, it may be that one computation requires 3 processors and the other requires only 1. This possibility indicates that such a scheduling algorithm will prolong the execution time and should not be used by a real operating system. The notion of scheduling and how it is actually implemented will be discussed in the following sections.

2.7 Minimum Spanning Tree

In this section we consider a more complex problem and show how it's parallel solution is coded as a ParC program. For a given undirected graph G with distinct integer weights on its edges, the goal is to find a subset of the edges which is a minimum spanning tree (MST) of G. The algorithm is based on the fact that the minimum cost edge (v, u) of G (which is unique) is included in any MST of G. This is because for any MST T of G that does not include (u, v) we can add (u, v) back to T and by so creates at least one cycle. Now we can break these cycles by removing other edges with larger weights and obtain a new MST with a smaller weight than T. Many algorithms were developed for MST and here we show a simple generalization of the sequential algorithm that maintains a set of sub-MSTs organized where each sub-MST is organized as a star (all nodes of the sub-MST point to a common root). The algorithm merges stars by selecting a minimum cost edge that connects two stars.

For input connectivity matrix W, let $w(i, j)$ be the weight of the edge connecting between nodes i and j and $w(i, i) = infinity$ otherwise. The algorithm's description is as follows:

1. Let $W^0 = W$ be the initial connectivity matrix, $n_0 = n$ initial number of stars (each node a star of size 1) and $k = 0$ the step counter.
2. While $|n^k > 1$ do:
3. $k = k + 1$;
4. For each node $v \in W^{k-1}$ do
5. Find an edge (v, u) with minimal weight such that
6. $W^{k-1}(v, u) = \min_{(v,x)}\{W^{k-1}(v, x)\}$.
7. Add the edge (v, u) to the MST rooted at v.

8. Shrink each sub-tree to a star. This is done by performing "pointer-jumping"

$$while(p \longrightarrow next \neq star.root)\ v \longrightarrow next = v \longrightarrow next \longrightarrow next;$$

   until all nodes in every sub-tree points to the root of that sub-tree.

9. Let n^k be the current number of stars.
10. Update the connectivity matrix $W^k = W^{k-1}$ restricted to all the roots of the current set of stars.

After each iteration we should have a set of stars that is smaller at least by half. The algorithm will terminate after at most $o(\log(n))$ iterations when only one star remains.

```
/* MINIMUM SPANNING TREE */
/* Parallel implementation of Sollin's algorithm */

/* INPUT :
     The graph is connected, undirected with  SIZE  vertices.
     W is a weight matrix of the edges. The highest value of
       an edge, SUP, and SIZE can be replaced by any other
       values.
   OUTPUT :
     An mst matrix where mst(i,j)=1 if the edge (i,j)
     belongs to the mst */

#include <stdio.h>

#define SIZE 11  /* number of vertices - size of problem */
#define SUP 100000  /* supremum value for weights */
#define TRUE 1
#define FALSE 0
#define MIN(a,b) ((a) < (b) ? (a) : (b))  /* returns
                                             MIN(a, b) */

typedef int mat[SIZE+1][SIZE+1];
mat w,  /* initial weighted matrix */
    w1,  /* iteration weighted matrix */
    w2,  /* auxiliary weighted matrix */
    mst;  /* MST matrix indication of the edges */
int c[SIZE+1],  /* vector of the minimum adjacent vertices */
    rs[SIZE],  /* the root vertices of the stars */
    n=SIZE;  /* number of vertices */

main()
{
  int nk,nk1,  /* number of rooted stars */
```

```
    i,j,
    isstar;  /* if all vertices are in rooted stars */
parfor(int i=1;i<=n;i++)  /* initialize w with the top
                          value */
{
  parfor(int j=i;i<=n;j++)
    {  w[i][j] = w[j][i] = SUP;  }
}
initialize();


parfor(int i=1;i<=n;i++)  /* initialize w1 and w2 to w */
{
  parfor(int j=i;i<=n;j++)
   {  w1[i][j]=w1[j][i]=w2[i][j]=w2[j][i] = w[i][j];  }
}


nk=nk1 = n;
while (nk1 > 1) {  /* terminate when there is one star */
  /* creating the forest defined by c[] */
  parfor(int v=1;v<= nk;v++)
  {
    int x,u,a,b,i;
    x = w1[v][1];  u = 1;  /* x = 1st element in the v
                           row */
    for (i=1; i<=nk; i++)  /* find minimum of the row */
     if (w1[v][i] < x) {
   u = i;
   x = w1[v][i];
  }
    c[v] = u;
    for (a=1; a<=n; a++)  /* restoring the original edges */
     for (b=a+1; b<=n; b++)
    if (w[a][b] == x) {  /* from the original matrix */
      mst[a][b] = mst[b][a] = 1;  /* add (v,c(v)) to the
                                  MST */
      a=b = n; }  /* exit loop */
  }


  /* shrink to rooted stars */
  parfor(int i=1;i<= nk;i++)  /* untie knots */
  {
    if (c[c[c[i]]] == c[i]) c[c[c[i]]] = c[c[i]];
  }
  do  {  /* perform pointer jumping */
    parfor(int i=1;i<= nk;i++)
     { c[i] = c[c[i]]; }
```

```
  isstar = star(nk);  }
while (!isstar);  /* until all vertices in rooted stars */


nk = nk1;
nk1 = numrs(nk);  /* computing nk */
/* assigning serial numbers to the rooted stars */
parfor(int i=1;i<= nk;i++)
{
  parfor(int j=1;i<= nk1;j++)
  {
if (c[i] == rs[j])  {
  c[i] = j;
  j=nk1;  }  /* break; */
  }
}


/* reinitialize w2 (with half of its previous dimension)
   to ensure that no previous values remain from
   the previous iteration in the smaller matrix. */
parfor(int i=1;i<=nk1;i++)
{
  parfor(int j=i;j<=nk1;j++)
{  w2[i][j] = w2[j][i] = SUP;  }
}
/* computing Wk */
/* determining the matrix elements,  find minimum
connecting edges */
/* between any two stars represented by elements
in the matrix of the */
/* previous iteration */
parfor(int i=1;i<=nk1;i++)
{
 int j,u,v,x;
 for (j=i+1; j<=nk1; j++)  {
x = SUP;
for (u=1; u<=nk; u++)
 for (v=u+1; v<=nk; v++)
  if (c[u]==i && c[v]==j || c[u]==j && c[v]==i)  /*stars
                                                 are
                                                 connect-
                                                 ed*/
     x = MIN(x, w1[u][v]);
w2[j][i] = w2[i][j] = x;
  }
}
```

```
    parfor(int i=1;i<=nk1;i++)  /* copy w2 to w1 */
    {
      parfor(int j=i;j<=nk1;j++)
      {  w1[i][j]=w1[j][i] = w2[i][j];  }
    }


  }  /* while */

    /* print result */
    /* the edge (i,j) belongs to the mst if and
    only if mst(i,j)=1 */
  printf("\n\n");
  for (i=1; i<=n; i++)  {
    printf("%2d  ",i);
    for (j=1; j<=n; j++)
      printf("%d ",mst[i][j]);
    printf("\n");
  }
}

/*********************************************************/
initialize()
{
}

/*********************************************************/
star(nk)
int nk;
{
  int i,
      s[SIZE+1],  /* vector of star indication to each
                     vertex */
      ret = TRUE;  /* boolean return value of the function */
  parfor(int i=1;i<=nk;i++)
    {  s[i] = TRUE;  }  /* initialize the s vector to TRUE */

  parfor(int i=1;i<=nk;i++)
  {
    if (c[i] != c[c[i]])  /* i does not point to root of
                             a star */
      s[i]=s[c[i]]=s[c[c[i]]]=FALSE;
  }


  parfor(int i=1;i<=nk;i++)
    {  s[i] = s[c[i]]; }

  parfor(int i=1;i<=nk;i++)
```

```
  {  if (s[i] == 0)  /* return false if at least one vertex */
      ret = FALSE;  /* does not belong to a star */
  }

  return(ret);
}


/*********************************************************/
/* computing the number of rooted stars.                 */
/* parameters : nk - the number of stars in the previous
iteration                                                 */
/* which is the current number of vertices.              */
/*********************************************************/
numrs(nk)
int nk;
{
  int i,j,found,count;
  count=1;  /* current number of stars */
  rs[1]=c[1];
  for (i=2; i<=nk; i++)  {
    found = FALSE;
    for (j=1; j<=count; j++)
      if (c[i] == rs[j]) /* if i points to an existing root */
    { found = TRUE;  break; }  /* terminate search */
    if (found == FALSE)  { /* i points to a new root (c[i]) */
      count++;  /* increase number of current stars */
      rs[count] = c[i];  /* put new root in rs */
    }
  }
  return(count);
}
```

2.8 Exercises

1. Show how to transform lparfor, parblock using parfor. In other words, describe the transformations using the syntax of these constructs.
2. Can we use lparfor instead of the parfor in the code of Fig. 2.11?
3. Discuss the possibility of adding a "Light parblock" construct to *ParC*, and show how to implement it using lparfor.
4. What is the difference between the outputs of the following loops:

$$for(i=0;i<n;i++)\ \{if(i\%2==0)i++;\ else\ printf(\text{“}\%d\text{”},i);\}$$

$$\text{parfor}\ int\ i;0;n-1;1\ \{if(i\%2==0)i++;\ else\ printf(\text{“}\%d\text{”},i);\}$$

5. Input and output statements can performed in parallel if they are supported by the operating system. Assume that a parallel machine allows the parallel execution of input and output statements as if they were atomic instructions.

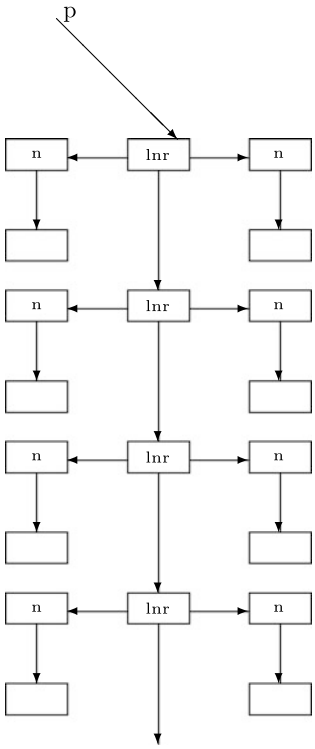
(a) What does the output of the following nested loops program look like?
(b) What is the effect of transforming the inner loop into a parallel loop?
(c) What is the effect of transforming the outer loop into a parallel loop?
(d) What is the effect of transforming both loops into a parallel loop?

```
int i,j;

for(i=0; i<10; i++) {
   for(j=0; j<i; j++)
      putchar('*');
   putchar('/n');
}
```

nested loops.

6. The following drawing describes a chain of n links with $5n$ nodes. Each one stores a number and has several pointer fields. The middle node of every link points to a left son and a right son, and to the middle node of the next link. Finally, each son points to a grandson, which points to null. Clearly, the chain can be accessed only through the pointer p.



(a) Write a non recursive parallel program that computes the sum of all of the nodes in this chain.
(b) Write a recursive parallel program that computes the sum of all of the nodes in this chain.
(c) Compute the parallel time (in steps) in which each of the above programs traverses the chain. What is the minimal parallel time of any such program?
(d) Describe a data structure that would allow us to compute the sum in $\log 5n$ steps (without losing the chain structure).
(e) Write a program that computes the sum of all of the values in the chain in less than $2n$ steps.
(f) Compare the number of processors needed by each of the above programs, such that the maximal degree of parallelism is obtained.

2.8.1 Questions About Summing

Three summing algorithms were presented: one recursive (R-SUM), one with an external loop (LP-SUM), and the third with an internal sequential loop (PL-SUM). The following set of questions is based on these sum variants:

1. Show that R-SUM is correct, by proving that all possible running sequences satisfy a certain invariant.
2. Based on the previous question, show that without the sync, PL-SUM is incorrect. This can be done by giving two possible execution orders for computing the sum of 8 numbers, with different sum results.
3. Consider two alternative ways for implementing the parallel prefix problem described earlier. The odd-even parallel prefix uses different strategies to divide the input. In parallel, it sums its n inputs in pairs, yielding $\frac{n}{2}$ new inputs, and recursively computes their parallel prefix. These $\frac{n}{2}$ partial sums fit with the partial sums of the even places of the original n inputs. The partial sums of the odd places are computed in parallel by adding every odd input to the partial sum of its even predecessor. This procedure is depicted on the left hand side of the network diagram in Fig. 2.34. An alternative way to compute partial sums is to construct a tree of sub-sums like the one created in LP-SUM. The correct partial sums are computed by propagating the sums of sub-trees from the root to the leaves. The rule for this back propagation assigns the left son the value received by its father. The right son is given the sum of the left son and the value received by its father. This back propagation of sub-tree sums is depicted on the left hand side of the diagram in Fig. 2.34.
   (a) Modify R-SUM such that it will implement the odd-even parallel prefix.
   (b) Modify R-SUM such that it will implement the tree-like parallel prefix.
   (c) Modify LP-sum such that it will implement the tree-like parallel prefix and use array cells to hold values stored at the tree nodes.

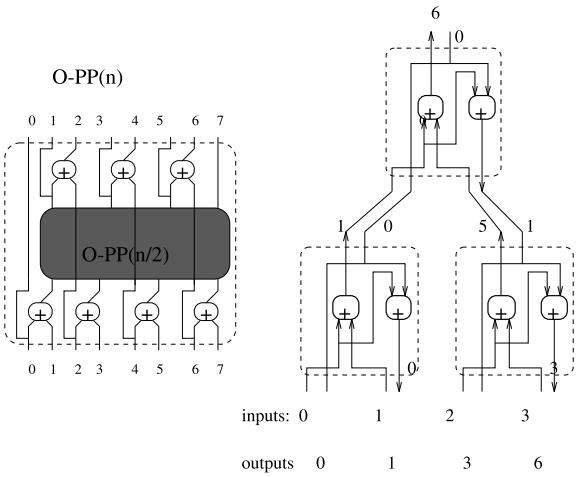


Fig. 2.34 Odd-even parallel prefix network and a tree-like parallel prefix network

(d) Compute the direct execution time for all three parallel prefix programs. Compare $N(R)$, $T(R)$ and the memory usage of all parallel prefix versions (including the version presented here).

(e) Create a more efficient version for each of the previous parallel prefix programs by taking into account that the number of processors is fixed and relatively small. $P < N$, meaning it is similar to the final version of PL-SUM.

4. Modify the parallel prefix operation from summing to ranking every non-zero cell in the array.
5. Modify the parallel prefix operation from summing to packing such that all non-zero elements of the array will be packed from left to right, preserving the initial order.
6. Change the following program so that all possible atomic effects can be described by different execution orders. This should be done by changing every external access to a local one (as described earlier). Remove superfluous assignments from your program. Modify the underlying program such that there will be only 10 possible execution orders. Is such a transformation useful as a general parallel programming technique?

```
PF i = 1 .. 10 [
    n = n -i;
    if((n+5) < (n*i)) n = n+5; else n = n-5;
]
```

7. Create a parallel version for $s2()$ in Sect. 7.8 where the first loop iterates until N, not $N/2$.
8. Show that the number of parallel steps performed by $p3()$ in Sect. 7.8 is optimal.

2.8.2 Questions About Min Algorithms

1. Program all five min algorithms described earlier, and test their correctness on the simulator.
2. For each of the five min programs, compute the expected execution time $T(R)$.
3. For each algorithm, determine the expected speedups $SP(R)$, $SP^o(R)$, $seq(R)$, $grain(R)$ and $glob_{bus}(R)$. Use the estimations given for the average number of iterations when needed (e.g., 6 in the case of random_min).
4. Test each algorithm on the simulator and compute all previous speedups for a specific N (large enough so that initialization at the beginning of the execution will not affect the results). Compare your calculations to the simulation results. Taking into account all of the factors, which algorithm is the best?
5. Justify the need for the second *pparfor* in *divide_crush*(). Explain why a direct substitution $ar[i] = crash(\ldots)$ is harmful.
6. Give an example showing that the *sync_min*() program can print an erroneous minimum if the external-while and the second *pparfor* are omitted.
7. Can the second *pparfor* in *sync_min*() be joined inside the first one?
8. Implement a version of *sync_min*() which uses *faa* instead of the external-while and the second *pparfor*.

2.8.3 Questions About Snake Sorting

Input: a two dimensional array A of size $N \times N$ of integers. The array in row order, if the rows are sorted in a snake-like order, will alternate in direction, with the first row going from left to right, the second row going from right to left and so forth. The last element in a row will be bigger than the first element of the next row. The sort algorithm that will be used throughout this test works as follows:

snake-sort(A)
LOOP Do $\log N$ times:
A Sort odd numbered rows to the right and even numbered rows to the left.
B Sort the columns (all of them downwards).
ELOOP

Assume that the sequential sort routines that the snake-sort uses to sort the rows and the columns are given, so you do not have to program them. Each one sorts a vector of K elements of A using $K \log K$ comparison steps:

l2r-row-sort($L, I, I + K$): Sort from left to right row L of A from I to $I + K$.
r2l-row-sort($L, I, I + K$): Sort from right to left row L of A from I to $I + K$.
col-sort($C, I, I + K$): Sort from bottom up column C of A from I to $I + K$.

All of these routines use a routine called “comp” that compares X and Y and sets the results as follows: NX to $max(X, Y)$ and NY to $min(X, Y)$.

```
comp(X,Y,NX,NY)
{
     if(X > Y) { NX = X; NY = Y;}
     else {NX = Y; NY = X}
}
```

Note that you should write your programs in ParC in free style, which means avoiding the "small" details of proper *C* programs. The program is acceptable as long as it does not ignore major complexities or aspects of the correct and full program. For example, in *comp*(...), we skipped parameter definitions and address references.

1. Use the zero-one principle to show that after $\log N$ iterations, the array is sorted by rows (show that the algorithm sorts an array of zeros and ones).
2. Write a program in ParC style using no more than 15 lines that executes the snake-sort algorithm using the above routines. Compute the speedup of this program ($P < N$) in all three models: without overhead, a simple overhead model and a general overhead model. Each of the sequential sorts routines that you use takes $N \log N$ steps of *comp*(...), so you have to consider those steps as well. In other words, count the steps of *comp*(...). Note that you should count condition-expressions and function-calls as statements.

   For the simple overhead model: Determine the value of Z' (the length of the longest path in the program). Substitute the value of all parameters (S, N, Z') in the speedup bounds equation given in class. Are the above bounds close to the exact speedup you found in the first part?

   The snake-sort algorithm "assumes" that $P = N$. This assumption is reflected in the fact that at any given time there are always N processes. Use the chancing method for sorting algorithms (Budget and Stevenson) for the case where $P < N$. Recall that chancing for sorting means replacing every comparison with a sequential merge routine that merges to "big blocks" of $\frac{N}{P}$ numbers each. Modify the snake-sort program to spawn no more than P processes. Modify the *comp*(...) routine to "merge" "big blocks." The modification of *comp*(...) should be done by using a sequential version of the snake-sort algorithm. Before listing the modified program, indicate:

   - What is the division of A into blocks?
   - What is the merge step of two blocks?

3. Modify the snake-sort algorithm to support the case of $P = \sqrt{N}N$ efficiently. Hint: The extra processors should be used to parallel the sequential sort of rows and columns.

   - What is the time needed for this case?
   - Using the above modification, describe a new modification for the case of $P = N^2$. Give the time equation for this case (you do not have to solve it, in case it is not straight forward).

4. Assume that you can partition A between the local memories of the processors, and assume that you can map processes to processors. For the case $P = N$ and a

buss machine there is extra memory overhead for every access of a processes to a value which is not in his local memory.

- What is a reasonable memory overhead for a bus machine? (recall that only one processor can use the bus at a time).
- Describe a good partitioning of A to local memories and the processes mapping to processors (for the program of Q1).
- Compute the speedup using the simple overhead model and the extra overhead criteria you found before.

5. Modify your snake-sort program of Q1 such that it will spawn only N processes in the beginning (in a manner similar to the transformation from LP-SUM to PL-SUM). The program should be valid now only for a synchronized execution. Assume that you have a sync command, and that the *comp*(...) operation is atomic.

   - If you iterate sync-snake-sort until A is in row order, will the program stop?
   - Indicate what else is needed in order to make sync-snake-sort run asynchronously. Recall the example of an asynchronous algorithm for the transitive closure of a graph.

2.9 Bibliographic Notes

Details regarding the C programming language can be found in Merrow and Henson (1989). The simulator is described in Ben-Asher and Haber (1995). There are set of libraries for emulating various hardware features for the simulator as Shaibe (1989) and Zernik and Rudolph (1991).

Nesting of parallel constructs that allows the user to express the spawning of a large number of threads can be found in Vrsalovic et al. (1989). A "strong model" for algorithms that computes the minimum (also sorting and merging) of n elements using p processors, can be found in Valiant (1975). Other parallel programing languages include: Unified Parallel C (UPC), which is an extension of the C programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory, see El-Ghazawi and Carlson (2005). High Performance Fortran (HPF) (Koelbel 1994) which is an extension of Fortran 90 with constructs that support parallel computing. Building on the array syntax introduced in Fortran 90, HPF uses a data parallel model of computation to support spreading the work of a single array computation over multiple processors. This allows efficient implementation on both SIMD and MIMD style architectures. HPF features included new Fortran statements, such as FORALL and the ability to create PURE procedures, compiler directives for recommended distributions of array data and additional library routines including environmental inquiry, parallel prefix/suffix, data scattering, and sorting operations. Titanium (Yelick et al. 1998)

is a high-performance Java dialect that includes parallel for each construct, multi-dimensional arrays that can be accessed in parallel, an explicitly parallel SPMD model a global address space through access to objects and objects fields. OpenMP (Dagum and Menon 2002) (Open Multi-Processing) is a common language to program shared memory machines (over C, C++ and Fortran). It include nesting of parallel for-loops and explicit control of scheduling policy. Unlike ParC its scoping rules allow only two modes of sharing: local variables and fully global variables. Finally, the CILK parallel programming language (Blumofe et al. 1995) is a parallel programing language that is based on spawn-thread construct rather than on the parfor construct. Spawning a thread is non-blocking and a special sync instructions is required to wait for spawned threads. Threads communicate through returned values that are collected by a sibling thread. Scheduling is done using Work-stealing paradigm where remote processors remove suspended threads from queues of other processors and execute them.

Scoping rules for nested parallel constructs as "cactus stack" implementation are presented in Hauck and Dent (1968), also described in Hummel and Schonberg (1991). More on scoping rules can be found in books dealing with programming languages as Scott (2009) and Sethi (1996). Scoping rules can be also dynamic as described in the books mentioned above, such that a global identifier refers to the identifier associated with the most recent environment (note that this cannot be done at compile time because the binding stack only exists at run-time). There are many works on obtaining atomic operations. Weihl (1990) supports atomic data-types by providing objects that provide serializability and recoverability of transactions on them. Eventually, guaranteeing atomicity yielded the transactional memory model wherein a sequence of instructions including load/store operations (code statements) in a parallel program is executed atomically. In transactional memory executing a code segment atomically does not imply exclusive lock over the full code but rather a weaker set of operations executed in a non-blocking way (Herlihy and Moss 1993; Marathe and Scott 2004).

Most of the algorithms on parallel sorting and maximum can be obtained from books on PRAM algorithms such as JáJá (1992) and Reif (1993). Some relevant and interesting lower bounds for problem discussed in this chapter can be obtained from Shiloach and Vishkin (1981). Alon and Spencer (1992) discuss Toran's theorem for Minimal Independence Set in graphs used to derive the lower-bound described at the end of this chapter. The parallel prefix is discussed in Snir (1986), Ladner and Fischer (1980) and in Hillis and Steele (1986).

Sollin's algorithm for Minimum Spanning Trees (MST) is presented in Sollin (1965). The first algorithm for finding a MST was developed by Boruvka in 1926 (see Nesetril et al. 2001). Commonly used algorithms that run in polynomial time Prim's and Kruskal's described in Cormen (2001). The fastest randomized algorithm for MST is Karger et al. (1995) and is based on Boruvka's algorithm and a reverse version of Kruskal's algorithm. While Chazelle's (2000) is the fastest deterministic algorithm and is close to linear time in the number of edges. Parallel algorithms for finding MST that are based on the Boruvka, Prim and Kruskal, do not scale well to additional processors. Chong et al. (2001) developed a parallel algorithm for

an EREW PRAM with linear number of processors, in optimal logarithmic time (a better randomized algorithm can be found in Pettie and Ramachandran 2004). Bader and Cong (2006) present a practical version for parallel MST with good speedup. Distributed algorithms for finding the MST were studied extensively (see Gallager et al. 1983; Awerbuch 1987; Garay et al. 2002). A lower bound of $\Omega(\frac{\sqrt{V}}{\log V})$ where v is the number of nodes was shown by Peleg and Rubinovich (2002).

References

Alon, N., Spencer, J.H.: The Probabilistic Method. Wiley, New York (1992)

Awerbuch, B.: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, pp. 230–240. ACM, New York (1987)

Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. J. Parallel Distrib. Comput. **66**(11), 1366–1378 (2006)

Ben-Asher, Y., Haber, G.: On the usage of simulators to detect inefficiency of parallel programs caused by bad schedulings: the simparc approach. In: HiPC (High Performance Computing), New Delhi, India (1995)

Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 207–216. ACM, New York (1995)

Chazelle, B.: A minimum spanning tree algorithm with inverse-Ackermann type complexity. J. ACM **47**(6), 1028–1047 (2000)

Chong, K.W., Han, Y., Lam, T.W.: Concurrent threads and optimal parallel minimum spanning trees algorithm. J. ACM **48**(2), 297–323 (2001)

Cormen, T.H.: Introduction to Algorithms. The MIT Press, Cambridge (2001)

Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (2002)

El-Ghazawi, T., Carlson, W.: UPC: Distributed Shared Memory Programming. Wiley-Interscience, New York (2005)

Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. **5**(1), 66–77 (1983)

Garay, J.A., Kutten, S., Peleg, D.: A sub-linear time distributed algorithm for minimum-weight spanning trees. In: Proceedings of 34th Annual Symposium on Foundations of Computer Science, 1993, pp. 659–668. IEEE, New York (2002)

Hauck, E.A., Dent, B.A.: Burroughs' B6500/B7500 stack mechanism. In: AFIPS Spring Joint Comput. Conf., vol. 32, pp. 245–251 (1968)

Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, p. 300. ACM, New York (1993)

Hillis, W.D., Steele, G.L. Jr.: Data parallel algorithms. Commun. ACM **29**(12), 1170–1183 (1986)

Hummel, S.F., Schonberg, E.: Low-overhead scheduling of nested parallelism. IBM J. Res. Dev. **35**(5/6), 743–765 (1991)

JáJá, J.: An Introduction to Parallel Algorithms. Addison Wesley Longman, Redwood City (1992)

Karger, D.R., Klein, P.N., Tarjan, R.E.: A randomized linear-time algorithm to find minimum spanning trees. J. ACM **42**(2), 321–328 (1995)

Koelbel, C.H.: The High Performance Fortran Handbook. The MIT Press, Cambridge (1994)

Ladner, R.E., Fischer, M.J.: Parallel prefix computation. J. ACM **27**(4), 831–838 (1980)

Marathe, V.J., Scott, M.L.: A qualitative survey of modern software transactional memory systems. Tech. Rep., University of Rochester Computer Science Dept. (2004)

Merrow, T., Henson, N.: System design for parallel computing. High Perform. Syst. **10**(1), 36–44 (1989)
Nesetril, J., Milková, E., Nesetrilová, H.: Otakar Boruvka on Minimum Spanning Tree Problem: Translation of Both the 1926 Papers, Comments. History. DMATH: Discrete Mathematics, vol. 233 (2001)
Peleg, D., Rubinovich, V.: A near-tight lower bound on the time complexity of distributed MST construction. In: 40th Annual Symposium on foundations of Computer Science, 1999, pp. 253–261. IEEE, New York (2002)
Pettie, S., Ramachandran, V.: A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. In: Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques, pp. 233–244 (2004)
Reif, J.H.: Synthesis of Parallel Algorithms. Morgan Kaufmann, San Francisco (1993)
Scott, M.L.: Programming Language Pragmatics, 3rd edn. Morgan Kaufmann, San Mateo (2009)
Sethi, R.: Programming Languages: Concepts & Constructs, 2nd edn. Pearson Education India, New Delhi (1996)
Shaibe, B.: Performance of cache memory in shared-bus multiprocessor architectures: an experimental study of conventional and multi-level designs. Master's thesis, Institute of Computer Science, The Hebrew University, Jerusalem (1989)
Shiloach, Y., Vishkin, U.: Finding the maximum, merging and sorting in a parallel computational model. J. Algorithms **2**(1), 88–102 (1981)
Snir, M.: Depth-size trade-offs for parallel prefix computation. J. Algorithms **7**(2), 185–201 (1986)
Sollin, M.: Le trace de canalisation. In: Programming, Games, and Transportation Networks (1965)
Valiant, L.G.: Parallelism in comparison problems. SIAM J. Comput. **4**(3), 348–355 (1975)
Vrsalovic, D., Segall, Z., Siewiorek, D., Gregoretti, F., Caplan, E., Fineman, C., Kravitz, S., Lehr, T., Russinovich, M.: MPC—multiprocessor C language for consistent abstract shared data type paradigms. In: Ann. Hawaii Intl. Conf. System Sciences, vol. I, pp. 171–180 (1989)
Weihl, W.E.: Linguistic support for atomic data types. ACM Trans. Program. Lang. Syst. **12**(2), 178–202 (1990)
Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., et al.: Titanium: A high-performance Java dialect. Concurrency **10**(11–13), 825–836 (1998)
Zernik, D., Rudolph, L.: Animating work and time for debugging parallel programs—foundation and experience. In: ACM ONR Workshop on Parallel and Distributed Debugging, pp. 46–56 (1991)

Springer

http://www.springer.com/978-1-4471-2163-3

Multicore Programming Using the ParC Language
Ben-Asher, Y.
2012, XIV, 277 p. 67 illus., Softcover
ISBN: 978-1-4471-2163-3