# SELSYN-C: A Self-Synchronizing Parallel Programming Language

Weiren Ding and Laurie J. Hendren *
School of Computer Science, McGill University
Montreal, Canada

**Abstract**

In this paper we report our design and implementation of a new self-scheduling parallel programming language, SELSYN-C. Our approach to the challenge of parallel programming language design and implementation is two-fold: (1) the design of simple extensions to C that are both easy to use, and useful for effective compilation, and (2) the design of efficient and effective scheduling strategies that can be automatically supported by a compiler and associated run-time environment.

## 1 Introduction and Motivation

The rapid advance of technology and architecture design is making possible rapid advances in the development of new and cheaper parallel processors. As these parallel processors become more accessible and more popular, the great challenge will be to provide software environments that make parallel processors usable by a wide variety of programmers. A critical step in this development is the design of parallel programming languages that are simple to use. Programs written in such a language should provide straight-forward mechanisms for expressing parallelism, and effective mechanisms for efficiently exploiting this parallelism on real parallel processors.

In this paper we present our new parallel programming language, SELSYN-C, an extension of the traditional C programming language that we have designed and implemented. We have selected the programming language C as the basis of our work because it is familiar to a wide range of programmers, and thus it provides a large user base for our new language. Users of our system need only master a small set of new constructs in order to develop parallel C programs, or to convert their existing C programs to parallel C programs. The two major contributions of this work are: (1) the development of simple-to-use extensions to C, and (2) the development of a compiler and associated run-time system that supports a new kind of self-synchronizing mechanism. As demonstrated by some examples, this approach provides a simple way to specify parallel programs, and an effective way to produce efficient programs that run on a shared memory parallel processor.

## 2 SELSYN-C, A Parallel C Language

One of our major goals was to design a language that is simple to use and familiar to a wide range of programmers. With this goal in mind, we decided to focus on designing simple extensions to the programming language C. These extensions were selected to be easy to use, but at the same time provide the necessary high-level information necessary for an effective translation to a real parallel processor. To keep these extensions portable for different target machines, our compiler performs a source-to-source transformation. Currently we have implemented the SELSYN system on BBN Butterfly GP-1000. With this implementation the programmer specifies his or her parallel program using SELSYN-C, and our compiler produces an output program in a C dialect that is specific to the BBN Butterfly. This output program is then linked with our run-time environment to produce a program that can be executed directly on the BBN Butterfly GP-1000.

SELSYN-C supports two major extensions to C: (1) the distinction between processor-private and globally-shared data, and (2) the introduction of weighted parallel function calls. We can illustrate these two extensions with the following example of a divide and conquer function to sum the entries in an array.

```
/* sum all entries a[l .. r], put in result */
sum(a,l,r,result)
int a[],l,r,*result;
{ shared int sum_left, sum_right;
  /* if only one entry left, then that is the sum */
  if (l == r)
     *result = a[l];
  else
    { /* sum left half and right half in parallel,
               result is sum of left and right */
      int midpoint;
      midpoint = (l + r) / 2;
      sum(a,l,midpoint, &sum_left) //
                  sum(a,midpoint+1,r,&sum_right);
      *result = sum_left + sum_right;
    }
}
```

As illustrated in the `sum()` example, the operator '//' indicates a parallel function call, and the keyword `shared` distinguishes between variables that are processor-private and globally shared. Note that it is only those variables whose address is used in parallel functions calls that must be declared as shared, all others are declared as in ordinary C programs. In general, there are the following two forms for parallel function calls:

$$p1(a_1, a_2, \ldots, a_n)//p2(a_1, a_2, \ldots, a_m)$$
$$p1(a1, a2, \ldots, a_n)@weight1//p2(a_1, a_2, \ldots, a_m)@weight2$$

The second type of parallel function call allows the programmer to add some information about the relative weights of the two procedure calls. These weights can be fixed constants, or they can be other expressions that

*This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

are evaluated at run-time to estimate the relative importance(weight) of the two function calls.

## 3 A Self-scheduling Mechanism

As we discussed in the introduction, the definition of a parallel language is not enough - we also need a way of effectively mapping the high-level parallel programs onto a real parallel machine. In order to solve this problem we have developed the SELSYN scheduling mechanism that is based on processor teams and a new idea of processor team cooperation. Our SELSYN-C compiler automatically inserts all of the necessary synchronization needed to support this scheduling mechanism, and thus the name of our language, **SEL**f-**SYN**chronizing **C**.

One possible implementation of parallel function calls is to use the traditional fork-join model implementation method in which a parallel function call of the form `f1() // f2()` is handled by forking the call `f2()` to a new processor and then joining again after the parallel function call. This solution, however, faces two problems on most shared memory multiprocessors, such as the BBN Butterfly. One problem is that the acquisition of a processor is a very expensive proposition [1]. If a heavy cost must be paid to acquire a processor, the granularity of the work in a fork-join block must be very large if the code is to run efficiently. The other major problem is that there is no performance advantage in having more runnable threads than available processors. Instead, this situation represents a performance liability, since the additional threads imply increased scheduler overhead.

### 3.1 The Processor-Team Model

Instead of incurring the large penalty of acquiring processors as the program executes, we have designed a mechanism that reduces processor acquisition costs through the use of processor-teams. When a program starts to execute it is assigned a team of processors that will be available to handle new tasks as they become available through the execution of parallel function calls. When a parallel function call such as `f1() // f2()` is encountered, the team is divided into two independent sub-teams, one sub-team will execute `f1()` and the other sub-team will execute `f2()`. This dividing procedure can be applied repeatedly as the problem is recursively decomposed until there is only one processor in each team.

A very simple strategy for team-dividing is illustrated in Figure 1. In this figure, the numbers enclosed in the "[ ]" represent the team members, and the numbers inside the node represent the processor in charge of the team. At the root node (corresponding to initiating the main program), there is only one team consisting of all processors which will take part in this computation (in this case processors 0 through 7). The main program begins executing on processor 0 and when a parallel function call is encountered, the team is divided into two equal size subteams, one subteam for each parallel function call. This dividing procedure continues for each parallel function call encountered until there is only one processor per team. When there is only one processor in a team there can be no further subdividing and subsequent parallel function calls are executed sequentially. In this manner, the overhead required to start up a new parallel task is only incurred when there are processors available for the task.

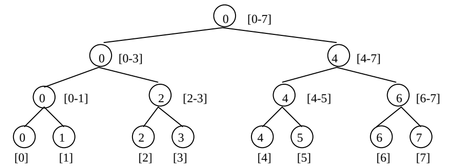


Figure 1: A Simple Form of Team Dividing

Note that in the processor-team model, the parallelism is exploited by rearranging the processors which is different than the process of acquiring new processors as is done in the traditional fork-join model. In addition, since the sub-teams are independent and controlled by different processors, the amount of contention and synchronization is reduced.

### 3.2 A Weighted Team-Division Strategy

In our previous `sum` example the workload in the two recursive procedure calls was equal, and thus evenly dividing the team into two equally sized sub-teams is clearly a good strategy. However, for other divide-and-conquer problems the workloads for the two recursive calls may vary widely, and the even-team-division mechanism will result in wasted processor resources. Now recall that our parallel function call mechanism provides the programmer with a way of specifying a weight associated with each function call. These weights can be used to express any sort of application specific information and may be dynamically evaluated at run-time. Examples of weights include: constants, expressions based on expected run-time complexity, or even expressions that calculate complex heuristics. Based on these programmer-supplied weights our weighted team-division mechanism calculates the sizes of the sub-teams based on: (1) the number of processors in the original team, and (2) the ratio of weight1 to weight2.[1]

### 3.3 Team-Cooperation

Although the dynamic strategy based on weights leads to improved processor utilization, we still need to deal with the problem of unbalanced work load distribution. This situation may occur when the programmer has given no weight information, or when the weight information does not always accurately predict the execution time. Thus, at run-time we may find the situation whereby one team has only one processor and has parallel tasks to be executed, while its sibling team may have one or more processors that are idle with nothing to work on. This situation clearly causes wasted processor resources, and we would like an inexpensive mechanism to help rebalance the workload when such a situation exists. Our solution is a new scheduling mechanism that we call team-cooperation. In order to explain our this mechanism, we introduce the following concepts:

[1] If no weights are given, then the division defaults to the simple strategy of dividing the team equally.

**Team:** A collection of processors. If there is more than one processor, a team can be divided into two sub-teams that work on independent parallel tasks. For example, in Figure 1, at the beginning there is only one team, team [0 - 7], and then this team is subdivided into two sub-teams, team [0 - 3] and team [4 - 7].

**Sibling Team:** When a team is divided into two sub-teams, each sub-team is the sibling team of the other. For example, in Figure 1, the team [0 - 3] and the team [4 - 7] are the sibling teams of each other.

**Leading processor:** The processor which has the lowest processor number in a team. This is the only processor which can issue the parallel tasks. In Figure 1, the numbers in the nodes represent the leading processors.

**Task Pool:** A pool may contain waiting tasks. It keeps the status of the tasks. Each processor has a task pool.

In the new team-cooperation model, cooperation is carried out between the leading processors of the sibling teams. In the simple team-division model a 1-processor team running on leading processor $P$ simply executes the two parallel tasks sequentially. However, in the team-cooperation model, processor $P$ takes one task to perform itself, and puts a concise description of the other task on its own task pool. This task is now available to processor $P$'s sibling team. If the leading processor of $P$'s sibling team, call it $P_{sib}$, finds itself idle waiting for $P$ to complete, then it will look in $P$'s task pool to see if there is a task to steal. If such a task exists then $P_{sib}$ steals it from $P$'s pool and executes the task. Note that the stolen task may execute new parallel function calls, and if the team associated with $P_{sib}$ has more than 1 processor, then the stolen task will be further sub-divided and executed on sub-teams associated with $P_{sib}$. When $P$ finishes its first task, it checks to see if the other task has been "stolen" by its sibling team's leading processor. If the task remains (the sibling team was always busy with its own tasks), $P$ will take back the task and execute it. Otherwise, $P$ can deduce that $P_{sib}$ has stolen the task, and perhaps has generated more subtasks. Thus, $P$ checks the task pool of $P_{sib}$ to see if it can steal a sub-task back. The full implementation details for this mechanism are rather complex, and are outside of the scope of this paper. The interested reader can refer to [4].

# 4 Experimental Results

In this section, we present the performance figures for the three scheduling strategies which have been described in the previous section. As indicated by the following pseudo-code, our test program is a version of quicksort that sorts linked lists. For our purposes, the important characteristic of this algorithm is that the execution time for the two parallel calls to quicksort may vary widely because of the different data amount. This unevenness of the partitioning step tests the ability of our mechanisms to deal with an unbalanced work load distribution which is unpredictable at compile-time.

```
quick-sort(head)
{ if (length(head) > 1) {
    /* partition using first element as pivot */
    partition(head, &part1, &part2, &part3);
    /* sort part1 and part3 in parallel */
    quick-sort(part1)@(length(part1))
      // quick-sort(part3)@(length(part3));
    combine-lists(part1, part2, part3);
  }
}
```

In Figure 2, we present the result of sorting 4096 elements. The x-axis represents the number of processors used for the test, while the y-axis represents execution time in machine ticks (one tick equals to 62.5 microseconds). The three lines give the performance for the Even-Team, Weighted-Team, and Cooperating-Team scheduling mechanisms. For this experiment both the Weighted-Team and Coordinated-Team tests used a parallel function call with weights. In particular, the weight for each recursive call to `quick-sort()` was given as the number of the elements in the input list.

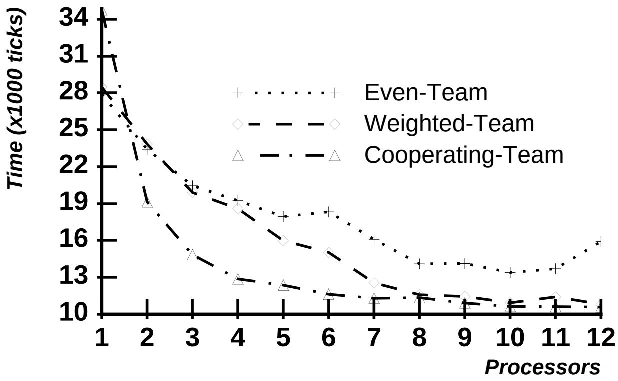


Figure 2: Quicksort: 4096 elements

Comparing the three performance lines, we can see that if the number of processors is greater than 2, the Cooperating-Team provides the best performance, the Weighted-Team is in second position and the Even-Team gives the worst performance. Note that the gap between the Even-Team and Weighted-Team lines indicates the improvement due to weights, and the gap between the Weighted-Team and Coordinated-Team lines indicates the further improvement due to the team-coordination mechanism.

Running the Even-Team on one processor is equivalent to running the sequential program. Therefore, by studying the performance of each mechanism for the one processor case, we can determine the overhead of the Weighted-Team and the Cooperating-Team mechanisms by comparing them with the Even-Team case. Note that even though the overhead is significant, we can see the benefits of the more expensive mechanisms for all cases with more than one processor.[2] Other experiments which indicate similar results have been performed and are reported in [4]. It should be noted that for some cases in which the workload

[2]Note that we are concerned only with the relative performance of the various scheduling mechanisms. To fully analyze the shape of the performance curves we would need to consider other factors like memory contention and data locality.

is evenly split, the Even-Team mechanism is superior. In these cases neither the weighted-division nor the cooperation improves the performance of the scheduling, since there is an overhead associated with both the Weighted-Team and Cooperating-Team mechanisms.

## 5 Related Work

The development of parallel programming languages has attracted much attention from researchers. Some have worked on developing entirely new languages for parallelizing programs, such as Strand, PCN, SISAL and BLAZE. The development on extensions to existing languages includes efforts on extensions to the language FORTRAN, such as EPEX/Fortran, Cedar Fortran and Force[7], includes developing extensions to language C, for instance, Concurrent C [3], Jade [10], PCP [2], and parallel-C [9], and includes efforts in the domain of symbolic languages like Lisp, such as Mul-T [8] and Multilisp [6]. Compared to these developments, our C extensions has the advantage of providing a large user base, being easy to use and understand. We have concentrated on minimizing the number of new keywords and operators, and we have introduced application specific extensions that are useful for the compiler (weights).

Backing up our language extensions, we developed a self-scheduling mechanism that can achieve high efficiency and good utilization of processors. Our work on scheduling is influenced by the work of WorkCrew [11] and PCP [2]. The WorkCrew approach introduces a mechanism that handles task divisions properly. Under this model, task subdivisions are queued for execution by any processor which becomes available while the requesting processor is busy. The WorkCrew approach won't divide a task unless there is a processor becomes idle and it will issue subtask on its own processor. However, it does pay a penalty for processor acquisition. Eugene Brooks introduced a split-join model rather than a fork-join model in [2]. In the split-join model the job starts out fully parallel with all of the processors it will ever have, and this team of processors is disassociated into independent subteams as nested concurrency is encountered. The weak points of this system is that it's lack of the ability to balance the work load between independent processors. Our mechanism has the advantage of both these two mechanisms. We reduce the cost by adopting the split-join model (which we call team-division), while at the same time we introduced team-cooperation to rebalance the work distribution. Rather than having a centralized task-pool, we create individual controller for each team so that the cost could be reduced further. Other related research on task scheduling has been introduced for Strand [5] and for Mul-T [8]. Our mechanism uses some similar ideas, but implemented in the domain of the popular language C.

## 6 Conclusions

This paper has reported on the SELSYN-C compiler and associated team-cooperation scheduling mechanisms. The SELSYN-C compiler has been implemented and is currently producing parallel code for a BBN GP-1000 parallel processor.

As discussed in this paper, the design of our the SELSYN-C language was driven by the need for an "easy-to-use" language that was accessible to a wide variety of programmers. Our SELSYN-C language consists of simple extensions to C that free the programmer from dealing with architectural details like processor management. Rather, the programmer can concentrate on the application and leave the details of scheduling and resource management to be handled completely by the the SELSYN-C compiler and associated run-time library. In order to provide an effective and efficient scheduling strategy we have incorporated the notion of weights into the SELSYN-C language, and we have developed a new team-cooperation scheduling mechanism that can make use of these weights. Finally, we have demonstrated the effectiveness of our approach on several divide-and-conquer applications.

## References


[1] E.D. Brooks III, T.S. Axelrod, and G.A. Darmohray. The cerberus multiprocessor simulator. In *Proc. Third SIAM Conference on Parallel Processing*, pages 1–4, December 1987.

[2] Eugene D. Brooks III. PCP: A parallel extension of C that is 99% fat free. In Tom MacDonald, editor, *Workshop on Scientific And Numerical Programming in C, ACM Supercomputing 89*, Novermber 1989.

[3] R. F. Cmelik, N. H. Gahani, and W. D. Roome. Experience with multiple processor versions of concurrent C. *IEEE Transaction on Software Engineering*, Vol. 15:335–344, March 1989.

[4] Weiren Ding. SELSYN-C: A self-synchronizing parallel programming language. Master's thesis, School of Computer Science, McGill University, May 1992.

[5] I. Foster and S. Taylor. Strand: New concepts in parallel programming. In Englewood Cliffs, editor, *Strand: New concepts in Parallel Programming*. NJ: Prentice-Hall, 1989.

[6] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[7] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Ruediger Jakdo. The force: A highly portable parallel programming language. In *Proceeding of the 1989 International Conference on Parallel Processing*, volume II, pages 112–117, 1989.

[8] D. A. Kranz, R. H. Halstead Jr., and E. Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings of the SIG-PLAN '89 Conference on Programming Languages Design and Implementation*, pages 81–90, 1989.

[9] James T. Kuehn and Howard Jay Siegel. Extensions to the C programming language for SIMD/MIMD parallelism. *IEEE*, pages 232–235, 1985.

[10] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proceeding of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, April 1991.

[11] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, Vol. 17(No. 4):347–366, 1988.