PCRC Runtime Library and HPF Compiler, *Extended abstract*

Bryan Carpenter, Geoffrey Fox, Xiaoming Li,
Xinying Li, Yuhong Wen and Guansong Zhang
Northeast Parallel Architectures Centre,
Syracuse University,
111 College Place,
Syracuse, New York 13244-410
{dbc,gcf,lxm,xli,wen,zgs}@npac.syr.edu

May 19, 1997

1 Introduction

The Web and Java have stolen some of the limelight over the past few years, but the role of parallel processing in the drive to ultimate computer performance remains unquestioned. National programs like Petaflops and ASCI attest continuing support for research and development in basic parallel processing technologies. While arguments will persist over what proportion of applications can benefit economically from parallel execution, we remain convinced that lack of good system software and tools is the main technical reason for the limited deployment of large scale parallel processing systems. Sponsored by ARPA, PCRC (Parallel Compiler Runtime Consortium) is an effort to establish a common base for parallel compiler development. This common base is a *runtime library*, which is supposed to be available to the compilers. It is linked to the generated "node program".

The basic idea is not new. Almost every compiler targetting a parallel architecture is supported by some sort of runtime, which resolves issues impossible to handle at compile-time. Those runtimes are usually designed at a low level in the hope of squeezing out the maximum performance. According to this mindset, low-level `send()` and `receive()` operations are encouraged in the node program. A consequence of this approach is that an *operational* parallel compiler is very difficult to construct—the compiler has too many details to take care of, and often ends up resorting to naive strategies. The result is that performance is much lower than it should be—promised global optimization by thin runtimes is hard to exploit. Our philosophy in PCRC is to build a relatively high level runtime library. We believe this approach will greatly facilitate compiler construction and deliver comparable or better performance for typical applications. An important aspect of PCRC runtime is that it is supposed to be a common base for different compilers. This demands a flexible design, and an implementation that adopts sound engineering practices.

PCRC is a joint effort among 10 organizations. This paper reports the work carried out at Syracuse University, in NPAC. In particular, section 2 introduces the architecture, functionality, and implementation of core PCRC runtime library (which will shortly be made public domain). Three groups of functions: data management, data movement, and computational routines, are described. Section 3 introduces an HPF compiler that uses this runtime. The compiler serves as an exemplar for the use of the runtime. This section also describes an infrastructure that has been developed in parallel with compiler construction. The algorithm for communication detection and the approaches employed for procedure interface processing and memory management are illustrated through segments of a node program generated by the compiler. Section 4 is a performance study. We compared performance of our compiler with a leading commercial HPF compiler on a Laplace solver and some test programs. The data suggest that our compiler produces higher performance and more consistent code than the commercial ones.

2 The NPAC run-time library

The kernel of NPAC library is a C++ class library. It is most directly descended from the run-time library of an earlier research implementation of HPF[1], with influences from the Fortran 90D run-time and the CHAOS/PARTI libraries. The kernel is currently implemented on top of MPI[2]. The library design is solidly object-oriented, but efficiency is maintained as a primary goal. Inlining is used extensively, and dynamic memory allocation, unnecessary copying, true procedure calls, virtual functions and other forms of indirection are generally avoided unless they have clear organizational or efficiency advantages.

The overall architecture of the library is illustrated in figure 1. At the top level there are several compiler-specific interfaces to a common run-time kernel. The four interfaces shown in the figure are illustrative. They include two different Fortran interfaces (used by different HPF compilers), a user-level C++ interface called ad++[3], and a proposed Java interface[4]. The development of several top-level interfaces has produced a robust kernel interface, on which we anticipate other language- and compiler- specific interfaces can be constructed relatively straightforwardly.

[1] Southampton HPF

[2] Because of the modular design of the library we anticipate that porting to other message-passing platforms would be relatively straightforward. The modules of the library that directly access the message-passing layer only contain around 10% of the total code. The only particularly MPI-specific feature used is MPI-style non-blocking communication.

[3] ad++ is currently implemented as a set of header files defining distributed arrays as type-secure container class templates, function templates for collective array operations, and macros for distributed control constructs.

[4] We also intend to produce a METACHAOS interface.

The largest part of the kernel is concerned with global communication and arithmetic operations on distributed arrays. These are represented on the right-hand side of figure 1. The communication operations supported include HPF/F90 array intrinsic operations such as `CSHIFT`, the function `writeHalo`, which updates ghost areas of a distributed array, the function `remap`, which is equivalent to a Fortran 90 array assignment between a conforming pair of sections of two arbitrarily distributed HPF arrays, and various gather- and scatter-type operations allowing irregular patterns of data access. Arithmetic operations supported include all F95 array reduction and matrix arithmetic operations, and HPF combining scatter. A complete set of HPF standard library functions is under development.

Nearly all these operations (including many of the arithmetic operations) are based on reusable schedules, in the PARTI/CHAOS mold. As well as supporting the inspector-executor compilation strategy, this organization is convenient in an object-oriented setting—a communication pattern becomes an object. As an illustration, consider the reduction operations. All reductions from a distributed array to a global result are described by an abstract base class using virtual functions for local block reductions. Specific instances such as `SUM` or `PRODUCT` are created by deriving concrete classes that instantiate the arithmetic virtual functions. This is a cleaner and more type-secure (hence, potentially, more efficiently compilable) alternative to passing function pointers to a generic reduction function.

For regular data movement operations a schedule consists of lists of source and destination blocks for local copies or send or receive operations. A block is defined as a multi-dimensional local array section parametrized by an offset and two short vectors of extents and strides. Where blocks are non-contiguous due to striding, or several blocks need to be communicated between the same pair of processors to execute a schedule, data is agglomerated by copying from user space to a buffer before sending, and copied back after receiving. Very long messages are split to avoid allocation of very large buffers by the library. Where a single contiguous block is to be sent between a pair of processes, no unnecessary copying is introduced. When local block copies involve contiguous data, a system operation such as `memcpy` is used.

All the data movement schedules are dependent on the infra-structure on the left-hand side of the figure 1. This provides the distributed array descriptor, and basic support for traversing distributed data ("distributed control"). Important substructures in the array descriptor are the *range* object, which describes the distribution of an array global index over a process dimension, and the *group* object, which describes the embedding of an array in the active processor set.

At the time of writing the kernel is fully functional and quite mature, two of the four interfaces illustrated are complete, and others are in progress.

3 Run-time based HPF compilation

The NPAC compiler is implemented as a translator from HPF to Fortran 77. It focusses on exploitation of explicit forall parallelism in the source HPF program. The translation scheme for forall assignments with linear subscripts is based on classification of the communication pattern needed in the construct. The "owner computes" rule is used to partition the computation, then the communication pattern is determined from the form of subscript expressions for arrays appearing on the right-hand side of the assignment. A compile-time *detect-communication* function recognizes three patterns:

- No communication: Corresponding array elements on the left-hand and right-hand sides of the assignment are located on the same processor.
- Shift communication: The right-hand side data need to be moved in parallel a constant distance along one or more dimensions. The *detect-communication* function returns that actual shift amount.
- Remap communication: The data movement is between different dimensions of the array, or the distance is not uniform along dimension.

The communication classification determines two aspects of the generated node program: memory allocation and execution control for computation and communication.

There are two memory allocation strategies used in our compiler: dynamically allocate a temporary for each RHS term, or allocate a "ghost area" for arrays that appear in RHS contexts where they need a small shift along the processor grid. The first method is used to handle "remap" communication. When a call to `pcrc_remap` is needed, a temporary array is allocated with the same alignment and distribution as the LHS target array and the RHS term is copied to the temporary array. The second method is used to efficently handle "shift" communication. If the compiler detects the need for a shift a "ghost area" is added to the RHS array. "Edge" elements are transferred using `pcrc_write_halo`. This saves the cost of copying a whole array.

The compiler must generate code and initialize the distributed array descriptors (DADs) passed to run-time functions and sub-programs. Using the PCRC-runtime Fortran interface, DAD initialization is straightforward.

As well as memory allocation, the node program must deal with translation between global array subscripts and local (node) subscripts. The run-time provides various functions to help with this translation[5]. The node program linearizes subscript computations for multi-dimensional arrays. Linearization of array segments, in conjunction with use of DAD inquiry functions provided in the runtime library, is important for implementing transcriptive features of HPF procedure, such as the `INHERIT` directive. Unnecessary copy-in and copy-out in caller or callee are generally avoided.

The HPF compiler is embedded in a compiler construction framework illustrated in the figure 2. Currently it handles various HPF language features, including data mapping directives, expressions and assignment, program execution control, I/O, and procedures.

The compiler construction framework, called the *frontEnd* system, is an extensible software package for constructing programming language processing tools. The system analyzes

[5] But these functions are never called from within the inner DO loops generated by translation of a forall.

the input source language and converts it to a uniform intermediate representation (IR) which can later be manipulated by a set of functions provided in a C++ class library.

The package is divided into the following components:

- **Language parsers** The language parsers are used to convert program source code to IR. The IR is designed to be suitable for multiple programming languages. We already have an HPF language parser (including a semantics checking module to check all constraints in the language definition) and a Java language parser. A C++ parser is being developed.

- **Package tools** These tools are used for displaying the intermediate representation. Two of them are available now. One tool unparses the IR back to the language source code for which the parse tree was generated. The other dumps the IR in plain text or HTML format for debugging purpose.

- **C++ class library** A C++ class library has been extended from the Sage++ class library. In the library, the nodes of the syntax tree are mapped to different C++ classes, with associated member functions provided to perform standard transformations, such as inserting or deleting statements from the tree.

- **Test suite** An extensive test suite is designed for different language parsers. For example, the package includes a "first order" exhaustive syntax test, in which each BNF syntax rule is used at least once. It also includes all example programs from "Migrating to Fortran 90" and the "HPF handbook" as test cases for the HPF parser.

The package is autoconfigured and tested on different platforms, including: IBM AIX, SUN OS/Solaris, DEC OSF, HP HP-UX, SGI IRIX and PC Linux.

4 Preliminary Benchmark Results

Figures 3 and 4 display select results of a benchmark comparision between the new NPAC compiler the PGI HPF compiler, version 2.0. The programs were run on the IBM SP2 at NPAC. The Laplace benchmark performs Jacobi relaxation on 256×256, 512×512 and 1024×1024 arrays, distributed blockwise over various numbers of processor. Both compilers achieve about the same performance on a single node, but generally our compiler exhibits better speedup on multiple processors, presumably due to more effective handling of communication. The synthetic benchmark involves no communication—it is a forall assignment involving large arrays. It illustrates that (unlike the PGI compiler) we deal with address translation efficiently, even for cyclic distribution format. (Speedup is relative to an equivalent sequential program compiled with the IBM Fortran compiler.)

While these examples are necessarily select, in general we find that (on code that both compilers can successfully compile) the NPAC compiler compares very favourably with the commercial compiler.

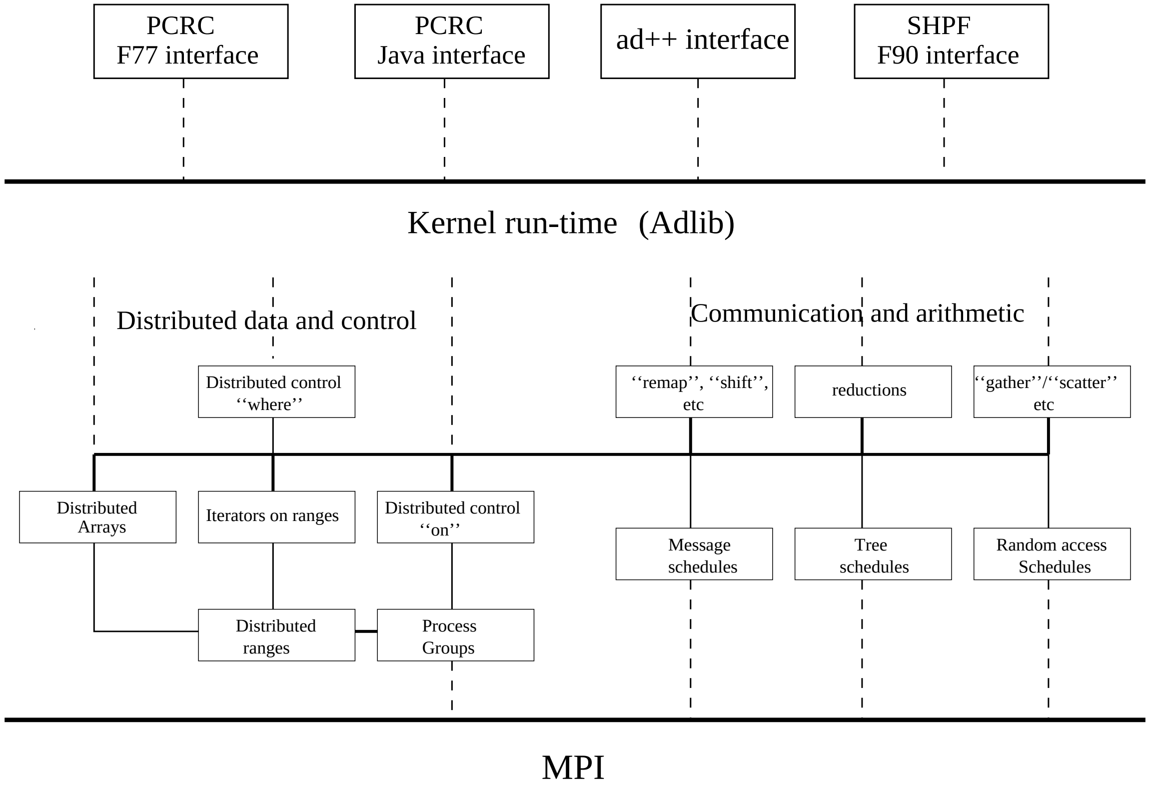

Figure 1: Architecture of the run-time.

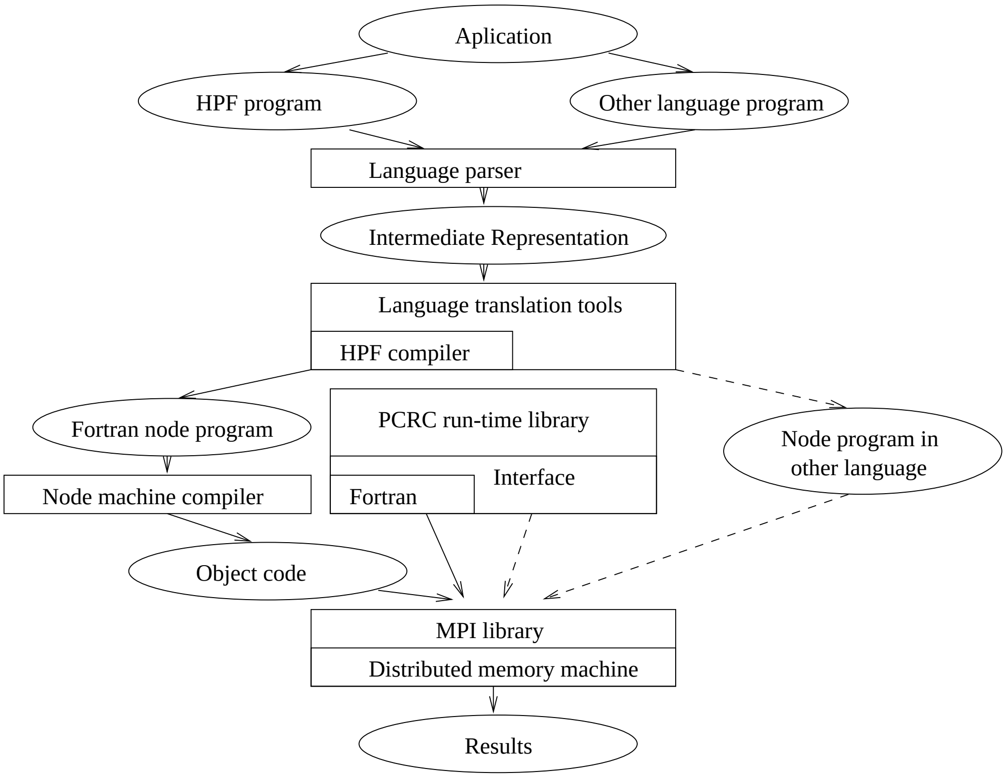

Figure 2: Compilation system overview

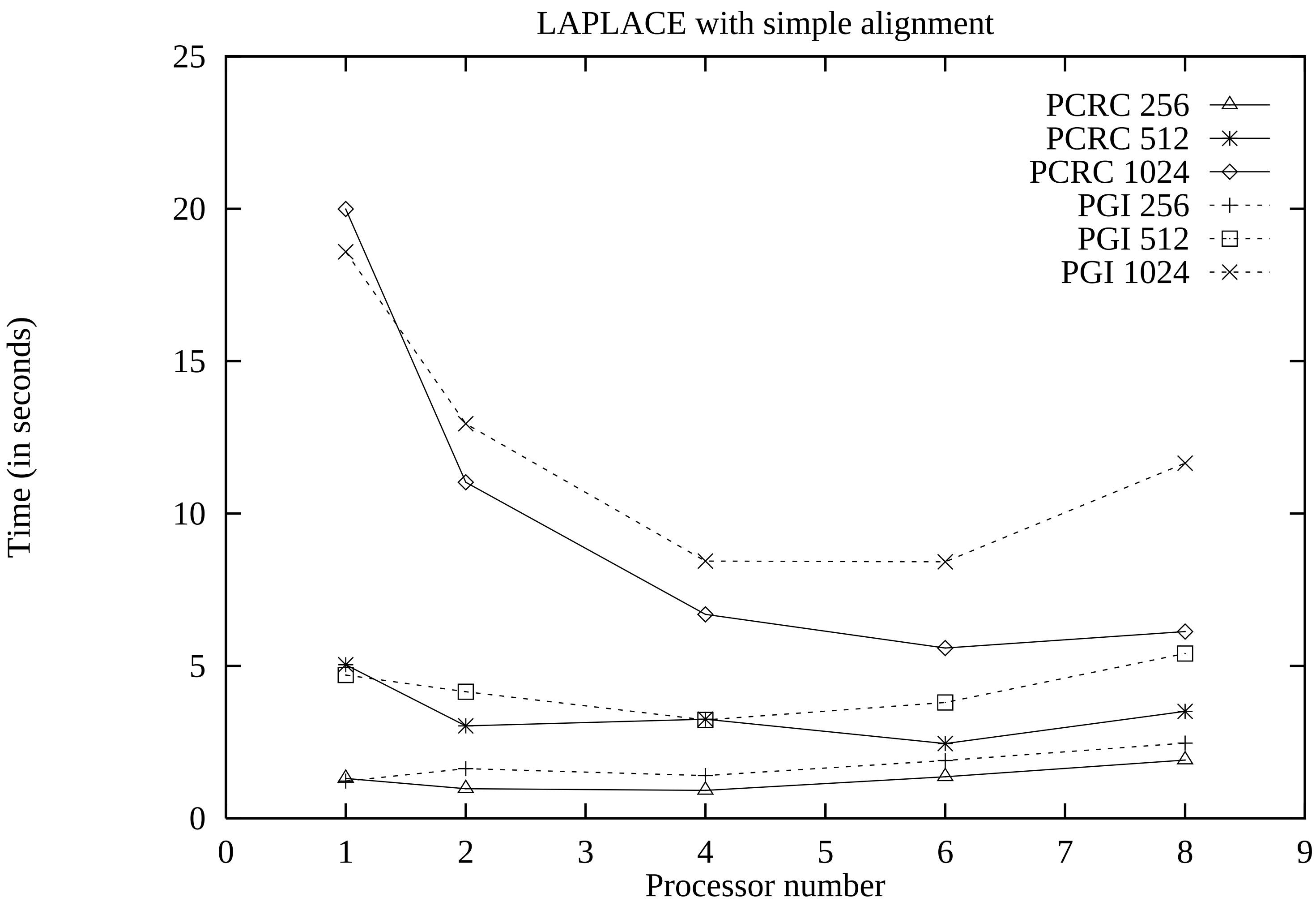

Figure 3: Laplace update.

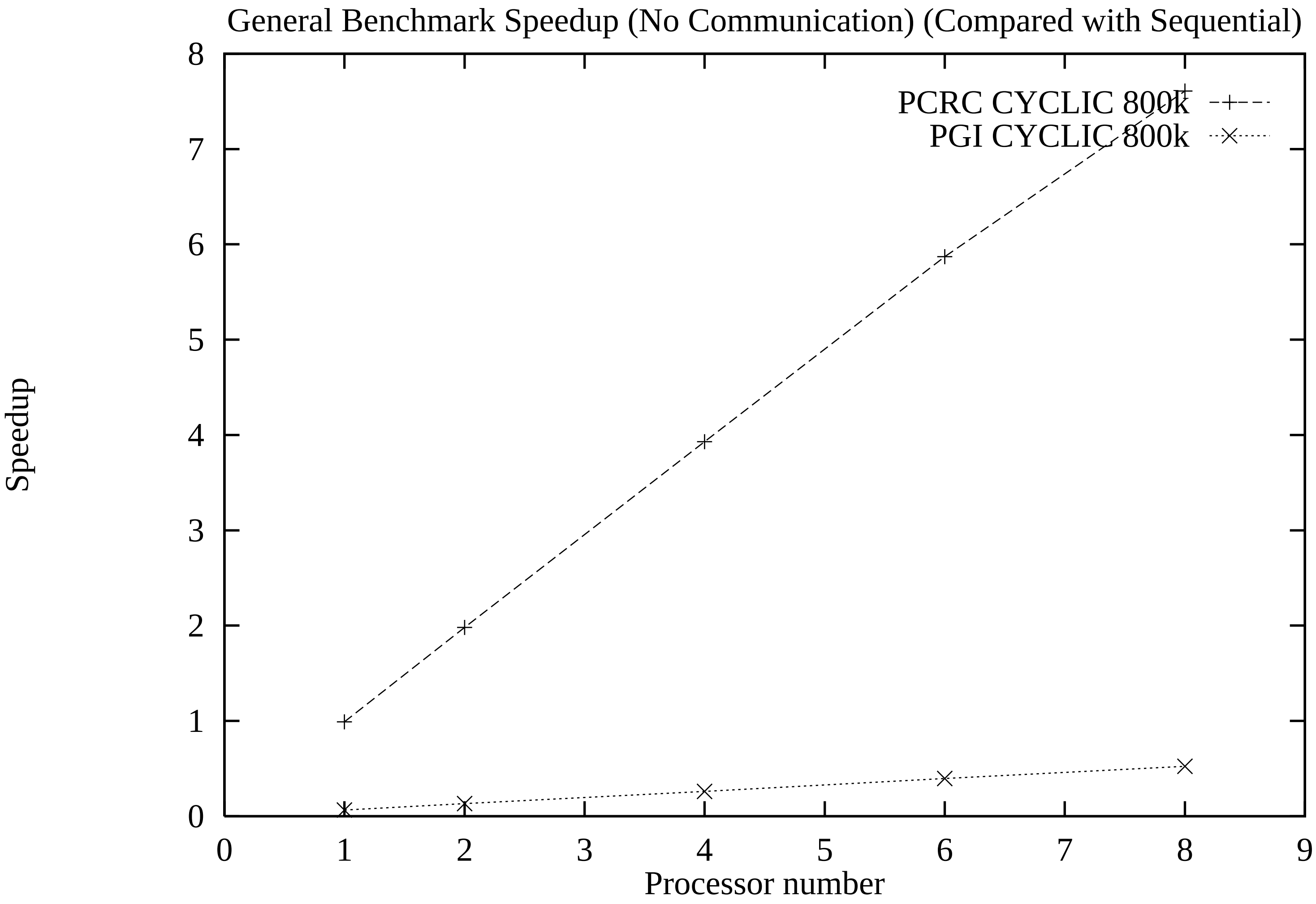

Figure 4: Synthetic benchmark involving cyclic distribution format.