To appear in: Proceedings of the 8th International Parallel Processing Symbosium (IPPS), Cancún, Mexico, April 1994.

Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers[1]

A. Malony, B. Mohr
Dept. of Comp. and Info. Sci.
University of Oregon
Eugene, Oregon 97403
{malony,mohr}@cs.uoregon.edu

P. Beckman, D. Gannon, S. Yang
Dept. of Comp. Sci.
Indiana University
Bloomington, Indiana 47405
{beckman,gannon,yang}@cs.indiana.edu

F. Bodin
Irisa
University of Rennes
Rennes, France
Francois.Bodin@irisa.fr

Abstract

pC++ is a language extension to C++ designed to allow programmers to compose distributed data structures with parallel execution semantics. These data structures are organized as "concurrent aggregate" collection classes which can be aligned and distributed over the memory hierarchy of a parallel machine in a manner consistent with the High Performance Fortran Forum (HPF) directives for Fortran 90. pC++ allows the user to write portable and efficient code which will run on a wide range of scalable parallel computers.

In this paper, we discuss the performance analysis of the pC++ programming system. We describe the performance tools developed and include scalability measurements for four benchmark programs: a "nearest neighbor" grid computation, a fast Poisson solver, and the "Embar" and "Sparse" codes from the NAS suite. In addition to speedup numbers, we present a detailed analysis highlighting performance issues at the language, runtime system, and target system levels.

1 Introduction

The introduction of a new parallel programming system should include, in addition to a description of the language principles and operational paradigm, an evaluation of the performance one would expect using the system, as well as a detailed accounting of the performance issues that have evolved from the system's design and implementation. However, as is often the case, the important concerns of portability, usability, and, recently, scalability of a parallel programming system tend to outweigh the equally important performance concerns when the system is released, leaving the mysteries of performance evaluation for users to discover. Certainly, the reasons for this situation are not hard to understand. The challenges of designing a language that supports a powerful parallel programming abstraction, developing a runtime system platform that is truly portable across diverse target hardware and software architectures, and implementing non-trivial applications with the system creates a large and complex software environment. Although the performance of the language, runtime system, and target system implementations are, clearly, always of concern during design and development, the time and effort needed to explore the performance ramifications of the initial versions of a parallel programming system may be difficult to justify if it delays system introduction.

However, the performance evaluation of a parallel programming system can be facilitated by integrating performance analysis support early in the system's design and development. This might occur in several ways, including:

- identifying performance events of interest at the language and runtime system levels;
- providing "hooks" for static and dynamic instrumentation; and
- defining execution abstractions that will be helpful when characterizing performance behavior.

The notion of *designing for performance analysis* is well-founded [22, 23], but until now has been rarely applied in the parallel language system domain.

The performance evaluation issues associated with the pC++ system are interesting because they address several performance levels (language, runtime system, target architecture) and require a system-integrated performance toolset to fully investigate. Hence, in concert with the pC++ system development, a performance analysis strategy has been formulated and is being implemented. As a result, the first version of the compiler — a preprocessor which generates Single Program Multiple Data (SPMD) C++ code that runs on the Thinking Machines CM-5, the Intel Paragon, the IBM SP-1, the BBN TC2000, KSR KSR-1, the Sequent Symmetry, and on a homogeneous cluster of UNIX workstations running PVM — is being introduced with integrated performance analysis capabilities and an extensive set of performance measurements already completed. These results are presented here.

[1] This research is supported by DARPA under Rome Labs contract AF 30602-92-C-0135.

The pC++ language and runtime system are very briefly described in §2 [1]. The performance measurement environment that is integrated in the pC++ system is described in §3. This environment is being used to perform a more detailed analysis of performance factors at the language, runtime system, and application levels. In §4, we describe four benchmark programs that we use to illustrate the performance issues associated with the pC++ language and runtime system implementation. Total execution time and speedup results are presented in §5. In §6, we present some of the detailed performance analysis results we have generated.

2 A Very Brief Introduction to pC++

The basic concept behind pC++ is the notion of a distributed collection, which is a type of concurrent aggregate "container class" [6, 8]. More specifically, a *collection* is a structured set of objects which are distributed across the processing elements of the computer in a manner designed to be completely consistent with HPF Fortran. To accomplish this, pC++ provides a very simple mechanism to build "collections of objects" from some base *element* class. Member functions from this element class can be applied to the entire collection (or a subset) in parallel. This mechanism provides the user with a clean interface to *data-parallel* style operations by simply calling member functions of the base class. In addition, there is a mechanism for encapsulating SPMD style computation in a thread based computing model that is both efficient and completely portable.

To help the programmer build collections, the pC++ language includes a library of standard collection classes that may be used (or subclassed). This includes classes such as *DistributedArray*, *DistributedMatrix*, *DistributedVector*, and *DistributedGrid*.

In its current form, pC++ is a very simple preprocessor that generates C++ code and machine independent calls to a portable runtime system. This is accomplished by using the Sage++ restructuring tools [3]. Sage++ is an object-oriented compiler preprocessor toolkit. It provides the functions necessary to read and restructure an internal representation of the pC++ program. After restructuring, the program is then "unparsed" back into C++ code, which can be compiled on the target architecture and linked with a runtime system specifically designed for that machine.

pC++ and its runtime system have been ported to several shared memory and distributed memory parallel systems, validating the system's goal of portability. The shared memory ports include the Sequent Symmetry [5], the BBN TC2000 [1], and the Kendall Square Research KSR-1 [2]. The distributed memory ports include the Intel Paragon [20], the TMC CM-5 [19], the IBM SP-1, and homogeneous clusters of UNIX workstations with PVM [24]. Work on porting the runtime system to the Cray T3D and Meiko CS-2 is in progress. More details about the pC++ language and runtime system can be found in [9, 10, 11, 12, 17].

[1] A companion paper, "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems", discusses issues of pC++ runtime system design and appeared in the Proceedings of the Supercomputing '93 conference [17].

3 The pC++ Performance Analysis Environment

The pC++ integrated performance analysis environment is unique because it is designed and implemented in concert with the pC++ language and runtime system. As a result of this tight coupling, the definition and analysis of performance factors is based in language and runtime execution semantics. However, this capability also presents a challenge to pC++ performance measurement since low-level performance instrumentation must be specified for capturing high-level execution abstractions, realized in performance measurements, and, finally, "translated" back to the application/language level. Presently the measurement environment consists of a profiling tool, a portable event trace capturing library, a source code instrumentor, and instrumented runtime system libraries. Analysis and visualization tools which use event trace data are under development; some are reported here. This section describes various aspects of the pC++ performance analysis environment.

3.1 Profiling pC++ Programs

In general, a very valuable tool for program tuning is function profiling. Simply, special instrumentation code is inserted at all entry and exit points of each function. This code captures data that can be used to calculate the number of times this function is called and the percentage of the total execution time spent in that routine. The necessary computations can be carried out in two ways: 1) the profile analysis can be directly performed at runtime (*direct profiling*), or 2) all entry and exit points of a function can be traced and calculations done off-line (*trace-based profiling*).

For pC++, we are interested in capturing performance profiling data associated with three general classes of functions: 1) thread-level functions, 2) collection class methods, and 3) runtime system routines. The data we want to capture includes activation counts, execution time, and, in the case of collections, referencing information.

3.1.1 General Approach

We perform all program transformations necessary for instrumentation at the language level, thus ensuring profiling portability. However, since profiling means inserting special code at all entry and exit points of a function, language-level profiling introduces the tricky problem of correctly instrumenting these points. In particular, we have to ensure that the exit profiling code is executed as late as possible before the function is exited. In general, a function can return an expression that can be arbitrarily complex, possibly taking a long time to execute. Correct profiling instrumentation would extract the expression from the return statement, compute its value, execute the profiling exit code, and finally return the expression result.

Luckily, we can let the C++ compiler do the dirty work. The trick is very simple: we declare a special *Profiler* class which only has a constructor and a destructor and no other methods. A variable of that class is then declared and instantiated in the first line of each function which has to be profiled as shown below for function `bar`.

```
class Profiler {
  char* name;
public:
  Profiler(char *n) {name=n; code_enter(n);}
  ~Profiler() {code_exit(name);}
};

void bar(){
  Profiler tr("bar"); // Profiler variable
  // body of bar
}
```

The variable is created and initialized each time the control flow reaches its definition (via the constructor) and destroyed on exit from its block (via the destructor). The C++ compiler is clever enough to rearrange the code and to insert calls to the destructor no matter how the scope is exited. Note also, that we use a private member to store a function identification which we can use in the destructor.

3.1.2 Profiling Implementation

The approach described above has two basic advantages. First, instrumenting at the source code level makes it very *portable*. Second, different implementations of the profiler can be easily created by providing different code for the constructor and destructor. This makes it very *flexible*. Currently, we have implemented two versions of the profiler:

Direct Profiling: The function profile is directly computed during program execution. We maintain a set of performance data values (`#numcalls`, `usec`, `cumusec`) for each profiled function. In addition, we store the current and parent function identifications and the timestamp of function entry in the `Profiler` object. These three values are set by the constructor, which also increments the counter `#numcalls`. The destructor uses the entry timestamp to compute the duration of the function call and adds this value to the corresponding `usec` and `cumusec` fields, but also subtracts it from the `usec` field of its parent function. In this way, we can compute the time spent in a function itself not counting its children. At the exit of the main function, all profile data gathered for all functions is written to a file by the destructor.

Trace-based Profiling: Here the constructor and destructor functions simply call an event logging function from the pC++ software event tracing library (see next subsection). All events inserted are assigned to the event class `EC_PROFILER`. By using event classes, the event recording can be activated/deactivated at runtime. The computation of the profile statistics is then done off-line.

Other profiling alternatives could be implemented in the same way. For example, profiling code could be activated/deactivated for each function separately, allowing dynamic profiling control. Another possibility is to let users supply function-specific profile code (specified by source code annotations or special class members with predefined names) that allows customized runtime performance analysis.

3.1.3 The pC++ Instrumentor

We use the Sage++ class library and restructuring toolkit to manipulate pC++ programs and insert the necessary profiler instrumentation code at the beginning of each function. The Instrumentor consists of three phases: 1) read the parsed internal representation of the program, 2) manipulate the program representation by adding profiling code according to an instrumentation command file, and 3) write the new program back to disk. Sage++ provides all the necessary support for this type of program restructuring.

By default, every function in the pC++ input files is profiled. However, the user can specify the set of functions to instrument with the help of an instrumentation command file. The file contains a sequence of instrumentation commands for including/excluding functions from the instrumentation process based on the file or class in which they are declared, or simply by their name. Filenames, classes, and functions can be specified as regular expressions.

3.1.4 The pC++ Runtime System

There are also instrumented versions of the pC++ class libraries and runtime system, both for direct and trace-based profiling. In addition to the instrumentation of user-level functions, they provide profiling of runtime system functions and collection access.

3.1.5 pprof and vpprof

Pprof is the parallel profile tool. It prints pC++ profile datafiles generated by programs compiled for direct profiling. The output of *pprof* is similar to the UNIX *prof* tool. In addition, it prints a function profile for each program thread and some data access statistics, showing the local and remote accesses to each collection per thread. Also, it prints a function profile summary (mean, minimum, maximum) and collection data access summary for the whole parallel execution. The function profile table has the following fields:

%time	The percentage of the total running time of the main program used by this function.
msec	The number of milliseconds used by this function alone.
total msec	A running sum of the time used by this function and all its children (functions which are called within the current function).
#calls	The number of times this function was invoked.
usec/call	The average number of microseconds spent in this function per call.
name	The name of the function.

Vpprof is a graphical viewer for pC++ profile datafiles. After compiling an application for profiling and running it, vpprof lets you browse through the function and collection profile data. It is a graphical frontend to *pprof* implemented using Tcl/Tk [15, 16]. The main window shows a summary of the function and the collection access profile data in the form of bar graphs. A mouse click on a bar graph object provides more detailed information.

3.2 Event Tracing of pC++ Programs

In addition to profiling, we have implemented an extensive system for tracing pC++ program events. Currently, tracing pC++ programs is restricted to shared-memory computers (e.g., Sequent Symmetry, BBN Butterfly, and Kendall Square KSR-1) and the uniprocessor UNIX version. The implementation of the event tracing package to distributed memory machines is under way[2]. Trace instrumentation support is similar to profiling. On top of the pC++ tracing system, we are implementing an integrated performance analysis and visualization environment. The performance results reported in this paper use utility tools to analyze the event traces that are based on externally available event trace analysis tools:

Merging: Traced pC++ programs produce an event log for each node. The trace files will have names of the form `<MachineId>.<NodeId>.trc`. The single node traces must be merged into one global event trace, with all event records sorted by increasing timestamps. This is done with the tool *se_merge*. If the target machine does not have a hardware global clock, *se_merge* will establish a global time reference for the event traces by correcting timestamps.

Trace Conversion: The utility tool *se_convert* converts traces to the SDDF format used with the *Pablo* performance analysis environment [18, 21] or to ALOG format used in the *Upshot* event display tool [4]. It also can produce a simple user-readable ASCII dump of the binary trace.

Trace Analysis and Visualization: The trace files can be processed with the *SIMPLE* event trace analysis environment or other tools based on the *TDL/POET* event trace interface [13, 14]. These tools use the Trace Description Language (TDL) output of the Instrumentor to access the trace files. In addition, we have implemented a Upshot-like event and state display tool (*offShoot*) based on Tcl/Tk [15, 16]. Like Upshot, it is based on the ALOG event trace format.

3.3 Programming Environment Tools

In addition to the performance tools, we started to implement some programming environment utilities. Currently, function, class, and static callgraph browsers are implemented. Future versions of pC++ will include data visualization and debugging tools.

4 Benchmark Test Suite

To evaluate the pC++ language and runtime system implementations, we have established a suite of benchmark programs that illustrate a wide range of execution behaviors, requiring different degrees of computation and communication. In this section, we describe the benchmark programs and point out features that make them particularly interesting for pC++ performance evaluation. The benchmarks were chosen to evaluate different features of the pC++ language and runtime system; two are related to CFD applications and two come from the NAS suite.

4.1 *Grid*: Block Grid CG

The first benchmark illustrates a "typical toy problem", grid computation. The computation consists of solving the Poisson equation on a two dimensional grid using finite difference operators and a simple conjugate gradient method without preconditioning.

[2]Note, the difference between the shared- and distributed-memory implementations is only in the low-level trace data collection library and timestamp generation; all trace instrumentation is the same.

Though this problem is very simple, it does illustrate important properties of the runtime system associated with an important class of computations.

In the program, we have used an early prototype of our *DistributedGrid* collection class. In addition, we have also used a common blocking technique that is often used to increase the computational granularity. The method used here is to make the grid size P by P and set the grid elements to be subgrids of size M by M; $M = \frac{N}{P}$.

The heart of the algorithm is a Conjugate Gradient iteration without any preconditioning. Communication occurs only in a function called Ap which applies the finite difference operator and in the dot product function, *dotprod*. In Ap, the communication is all based on nearest neighbors and in the *dotprod* function the communication is all based on tree reductions.

4.2 *Poisson*: Fast Poisson Solver

Another common method for solving PDEs is to use a fast Poisson solver. This method involves applying a Fast Fourier Transform based sine transform to each column of the array of data representing the right hand side of the equation. This is followed by solving tridiagonal systems of equations associated with each row. When this is complete, another sine transform of each column will yield the solution. In this case it is best to view the grid as a distributed vector of vectors, where the vector class has a special member function for computing the sine transform, $sineTransform()$.

The distributed vector collection must have a special function for the parallel solution of tridiagonal systems of equations. In our case we use a standard cyclic reduction scheme. This is accomplished by building a collection class *DistributedTridiagonal* which is a subclass of *DistributedVector*. This class has a public function *cyclicReduction*() which takes two parameters which correspond to the diagonal and off-diagonal elements of the matrix which are stored in each element.

The cyclic reduction function is organized as two phases of $log(n)$ parallel operations. The first phase is the forward elimination, the second is the back solve step. Communication only happens within the element function $forwardElimination$ and $backSolve$. The final poisson solver uses two collections F and U which represent the initial data and the final solution. Both are of type distributed tridiagonal of vector which is easily mapped to a one dimensional template.

4.3 *Embar*: NAS Embarrassingly Parallel Benchmark

The NAS benchmark suite was designed to test the suitability of massively parallel systems for the needs of NASA Ames Research. Of the nine codes in the suite, four have been translated to pC++ and we report on two of them here. (A more complete report on this suite will be published when all have been translated and tested.)

The easiest of these is the "Embarrassingly Parallel" program. This code generates 2^{24} complex pairs of uniform $(0, 1)$ random numbers and gathers a small number of statistics. In the benchmark, the random numbers are created by two special portable functions called $vranlc()$ and $randlc()$, and a main function *compute_pairs(int k)* is used to compute the numbers based on a different value of the seed parameter k. Our approach is to divide the work into a set of computational engines with one engine per processor. This "set" of engines is a collection of elements of type `Set<GaussianEngine>`. Each GaussianEngine object computes $\frac{1}{NODES}$ of the total where $NODES$ is the number of processors.

4.4 *Sparse*: NAS Sparse CG Benchmark

A far more interesting benchmark in the NAS suite is the random sparse conjugate gradient computation. This benchmark requires the repeated solution to $A * X = F$, where A is a random sparse matrix. There are two ways to approach the parallelization of this task. The best, and most difficult, is to view the matrix as the connectivity graph for the elements in the vector. By a preprocessing step one can partition the vector into segments of nearly equal size that minimizes the communication between subsets. The resulting algorithm can have the same low communication to computation ratio as our simple grid CG example above.

The second approach is not as efficient in terms of locality, but it a more direct implementation of the benchmark. In this version we represent the matrix as a distributed random sparse matrix. More specifically, we represent the matrix as a Distributed Matrix of Sparse Matrices. The $DistributedMatrix$ collection class in the pC++ library allows any class that has the algebraic structure of a ring to be the element of the matrix. Furthermore, a p by p matrix whose elements are k by k matrices has all the same mathematical properties of a kp by kp matrix. Indeed, this is the key property used in the blocking algorithms used in most BLAS level 3 libraries. Consequently, we can select the element class to be a sparse matrix. This distributed matrix of sparse matrices can then be used like an ordinary matrix.

5 Performance Scalability

A key feature of the pC++ programming system is its ability to transparently and efficiently support scaling of problem size and machine size. Clearly, the dominant performance concern is pC++'s ability to achieve scalable performance. As the problem size increases, pC++ should support increased processing efficiency. In response to increasing numbers of processors, pC++ should demonstrate good speedup. Scalability results for shared and distributed versions of the pC++ system are given below.[3] We also provide comparisons to Fortran results where appropriate.

5.1 Shared Memory

The shared memory ports of the pC++ runtime system isolate performance issues concerned with the allocation of collections in the shared memory hierarchy and architectural support for barrier synchronization. Clearly, the choice of collection distribution is important, but as is typically the case in programming shared memory machines, the ability to achieve good memory locality is critical.

The scaled performance of shared memory ports of the pC++ system reflects the effectiveness of collection distribution schemes as well as the interactions of the underlying architecture with respect to collection memory allocation, "remote" collection element access, and barrier synchronization. Figures 3, 4, and 5 show the speedup of the benchmark programs on the Sequent, BBN, and KSR machines, respectively.

Naturally, the speedup of Embar was excellent for all machines. For the Sequent using 23 processors, the speedup of 15.94 reflects the mismatch between the number of processors and the problem size; the staircase behavior is even more pronounced in the BBN results. The slightly lower speedup for Embar on 64 nodes of the KSR-1 is due to the activation of an additional level of the cache memory hierarchy; a portion of the memory references must cross between cluster rings, encountering latencies 3.5 times slower than references made within a 32 processor cluster. This effect is clearly evident for the Grid benchmark on the KSR, whereas the other machines show steady speedup improvement; the 40 to 60 processor cases on the BBN TC2000 are again due to load imbalances caused by the distribution not being well matched to that number of processors.

The Poisson benchmark performs well on all shared memory machines for all processor numbers. This demonstrates pC++'s ability to efficiently assign elements of a collection, such as the distributed vector collection, to processors and use subclassing to implement high-performance functions on the data, such as cyclic reduction.

The speedup performance on the Sparse benchmark reflects the importance of locality; most evident in the KSR results. The uniform memory system of the Symmetry hides many of the poor locality effects, resulting in a reasonable speedup profile. The NUMA memory system of the BBN TC2000 is more susceptible to locality of reference because of the cost of remote references. We knew that the Sparse implementation was not the most efficient in terms of locality, but this resulted in particularly poor speedup performance on the KSR-1; when crossing cluster boundaries, the drop is speedup is quite severe. Although the pC++ port to the KSR-1 was done most recently and should still be regarded as a prototype, the analysis of the performance interactions between collection design, distribution choice and the hierarchical, cache-based KSR-1 memory system is clearly important for optimization.

5.2 Distributed Memory

In contrast to shared memory ports of pC++, the principal performance factors for distributed memory versions of the runtime system are message communication latencies and barrier synchronization. Collection design and distribution choice are the major influences on the performance of an application, but runtime system implementation of message communication and barrier synchronization can play an important role. In fact, these factors affect performance quite differently on the TMC CM-5 and Intel Paragon. Communication latency is very low for the CM-5 and the machine has a fast global synchronization mechanism. In the Paragon, communication performance is poor, requiring parallelization schemes that minimize communication for good performance to be achieved.

Figures 1 and 2 show the execution time speedup results for the benchmark suite on the CM-5 and Paragon machines, respectively. The Embar benchmark shows excellent speedup behavior on both machines. For the CM-5, the execution time was within 10 percent of the published hand optimized Fortran results for this machine. In the case of the Paragon, a 32 node Paragon achieved a fraction of 0.71 of the Cray uniprocessor Fortran version; speedup was 19.6 with respect to this code.

The Grid benchmark demonstrated near linear speedup on the CM-5 even with 64 by 64 sub-blocks, reflecting the low communication overhead relative to sub-block computation time. However, the high communication overheads on the Paragon machine required a different block size choice, 128 instead of 64, before acceptable speedup performance could be achieved on this benchmark.

Execution time for Poisson is the sum of the time for FFT transforms and cyclic reduction. Because the transforms require no communication, their performance scales very well for both the CM-5 and the Paragon. In contrast, the cyclic reduction requires a communication complexity that is nearly equal to the computational complexity. Although the communication latency is very low for the CM-5, no speedup was

[3] The ports to the IBM SP-1 and workstation clusters using PVM were done recently. Performance results for this ports were not yet available for inclusion in this article.

observed in this section even for Poisson grid sizes of 2,048; because of the larger number of processors used, the communication to computation ration was high. With a smaller number of processors, the Paragon was able to find some speedup in the cyclic reduction part.

Finally, running the Sparse benchmark, the CM-5 achieved a reasonable speedup, and for 256 processors matched the performance of the Cray Y/MP untuned Fortran code. In the case of the Paragon, the intense communication required in the sparse matrix vector multiply, coupled with high communication latency, actually resulted in a slowdown in performance as processors were added. We cannot expect improvements in these numbers until Intel finishes their "performance release" of the system.

Currently, Indiana University is experimenting with Sandia's high performance SUNMOS as the compute-node operating system of its Intel Paragon. SUNMOS increases message passing bandwidth and reduces latency. For compute-bound benchmarks such as Embar, preliminary results show no significant improvement. However for communication intensive benchmarks such as Sparse, initial results show a factor of nearly 7 improvement over nodes running OSF/1.

6 Detailed Performance Analysis

The pC++ performance environment allows us to investigate interesting performance behavior in more detail. In particular, a pC++ program execution involves several execution components: initialization, processor object instantiation, thread creation, collection creation, runtime system operation, and the application's main algorithm. To associate performance data with these important operationally-semantic components, we formulated an instrumentation model that we then applied to detailed performance studies. This instrumentation model is represented in Figure 6. Essentially, we identify events at the begin and the end of the program as well as at the begin and end of each major execution component (event capture points are indicated by circled numbers in the figure). From these events, the following time measurements are computed:

(1)-(8) (*setup*) : The entire benchmark program.

(2)-(8) (*fork*) : The part of the program which runs in parallel, starting with the forking of processes.

(3)-(7) (*main*) : The main pC++ program as supplied by the user. It includes static collection allocation, user setup and wrapup, and the parallel algorithm.

(4)-(7) (*user*) : The computation part representing "pure" user code execution, without the overhead of collection allocation.

(5)-(6) (*parallel*) : The parallel algorithm portion of the pC++ program. The time in this section corresponds to the measurements reported in §5.

Our first application of the above instrumentation and measurement model was to determine, using tracing, the relative influence of different phases of the entire benchmark execution where the language and runtime system execution components were involved. Because the speedup results reported in §5 are only for the *parallel* section, we wanted to characterize the speedup behavior in other program regions. An example of the detailed performance information we are able to obtain is shown for the Poisson benchmark in Figures 7, 8, and 9 for the shared memory pC++ ports. In addition to the phases described above, the figures show also the speedup profile of the `sineTransform` (fft) and `cyclicReduction` (cyclic) functions as described in Section 4.2. The graphs clearly show how overall performance is degraded by components of the execution other than the main parallel algorithm, which scales quite nicely. Although some of these components will become relatively less important with scaled problem sizes, understanding where the inefficiencies lie in the pC++ execution system will allow us to concentrate optimization efforts in those areas.

We use profiling measurements as a way of obtaining more detailed performance data about runtime system functions, thread-level application functions, collection class methods, and collection referencing. The pC++ profiler and instrumentation tools allow different levels and types of performance information to be captured. Whereas the type of measurement above helps us identify pC++ system problems, a pC++ programmer may be particularly interested in information about where the parallel algorithms is spending the most time and its collection referencing behavior.

7 Conclusion

The pC++ programming system includes an integrated set of performance instrumentation, measurement, and analysis tools. With this support, we have been able to validate performance scalability claims of the language and characterize important performance factors of the runtime system ports during pC++ system development. As a consequence, the first version of the compiler is being introduced with an extensive set of performance experiments already documented. Some of the performance analysis has been reported in this paper. From the scalability data, we see that pC++ already achieves good performance. From the detailed trace and profile data we are able to pinpoint those aspects in the language's use for algorithm design or in the implementation of runtime system operations where performance optimizations are possible.

For instance, the profiler has shown that a great number of barrier synchronization are generated, causing a reduction in overall parallelism. One important compiler optimization will be to recognize when barriers can be removed or replaced with explicit synchronization. Other optimizations might be more architecture specific. As an example, in distributed memory systems it will be important to overlap communication with computation, whereas in shared memory environments, collection distribution and memory placement will be important for achieving good locality of reference. Again, performance analysis will be critical for identifying the need and resulting benefit of such optimizations.

For more information ...

Technical documents and the programs for pC++ and Sage++ are available via anonymous FTP from `moose.cs.indiana.edu` and `ftp.cica.indiana.edu` in the directory `~ftp/pub/sage`. We maintain two mailing lists for pC++/Sage++. For information about the mailing lists, and how to join one, please send mail to `sage-request@cica.indiana.edu`. No subject or body is required. Also, the pC++/Sage++ project has created a World-Wide-Web server. Use a WWW viewer to browse the on-line user's guides and papers, get programs, and even view pictures of the development team. The server can be found at `http://www.cica.indiana.edu/sage/home-page.html`.

References

[1] BBN Advanced Computer Inc., Cambridge, MA. *Inside the TC2000*, 1989.

[2] S. Frank, H. Burkhardt III, J. Rothnie, *The KSR1: Bridging the Gap Between Shared Memory and MPPs*, Proc. Compcon'93, San Francisco, 1993, pp. 285–294.

[3] D. Gannon, F. Bodin, S. Srinivas, N. Sundaresan, S. Narayana, *Sage++, An Object Oriented Toolkit for Program Transformations*, Technical Report, Dept. of Computer Science, Indiana University. 1993.

[4] V. Herrarte, E. Lusk, *Studying Parallel Program Behavior with Upshot*, Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[5] Sequent Computer Systems, Inc. *Symmetry Multiprocessor Architecture Overview*, 1992.

[6] A. Chien and W. Dally. *Concurrent Aggregates (CA)*, Proc. 2nd ACM Sigplan Symposium on Principles & Practice of Parallel Programming, Seattle, Washington, March, 1990.

[7] High Performance Fortran Forum, *High Performance Fortran Language Specification*, 1993. Available from titan.cs.rice.edu by anonymous ftp.

[8] J. K. Lee, *Object Oriented Parallel Programming Paradigms and Environments For Supercomputers*, Ph.D. Thesis, Indiana University, Bloomington, Indiana, Jun 1992.

[9] J. K. Lee, D. Gannon, *Object Oriented Parallel Programming: Experiments and Results*, Proc. Supercomputing 91, Albuquerque, IEEE Computer Society and ACM SIGARCH, 1991, pp. 273–282.

[10] D. Gannon, J. K. Lee, *Object Oriented Parallelism: pC++ Ideas and Experiments*, Proc. of 1991 Japan Society for Parallel Processing, pp. 13–23.

[11] D. Gannon, J. K. Lee, On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions, *Proc. CONPAR 92–VAPP*, Lyon, France, Sept. 1992.

[12] D. Gannon, *Libraries and Tools for Object Parallel Programming*, Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, St. Hilaire du Touvet, France, Elsevier, Advances in Parallel Computing, Vol. 6, pp. 231–246, 1993.

[13] B. Mohr, *Performance Evaluation of Parallel Programs in Parallel and Distributed Systems*, H. Burkhart (Eds.), Proc. CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing, Zurich, Lecture Notes in Computer Science 457, pp. 176–187, Berlin, Heidelberg, New York, London, Paris, Tokio, Springer Verlag, 1990.

[14] B. Mohr, *Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?*, Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, St. Hilaire du Touvet, France, Elsevier, Advances in Parallel Computing, Vol. 6, pp. 103–124, 1993.

[15] J. K. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. 1990 Winter USENIX Conference.

[16] J. K. Ousterhout, *An X11 Toolkit Based on the Tcl Language*, Proc. 1991 Winter USENIX Conference.

[17] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, *Implementing a Parallel C++ Runtime System for Scalable Parallel Systems*, Proc. 1993 Supercomputing Conference, Portland, Oregon, pp. 588–597, Nov. 1993.

[18] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Birkett, D. W. Jensen, B. A. A. Nazief, B. K. Totty, *Scalable Performance Environments for Parallel Systems*. Proc. 6th Distributed Memory Computing Conference, IEEE Computer Society Press, pp. 562–569, 1991.

[19] J. Palmer, G. L. Steele, Jr. *Connection Machine Model CM-5 System Overview*, Proc, 4th Symp. Frontiers of Massively Parallel Computation, pp. 474–483.

[20] Intel Supercomputes, *Paragon-XP/S Technical Specification.*, Beaverton, Or.

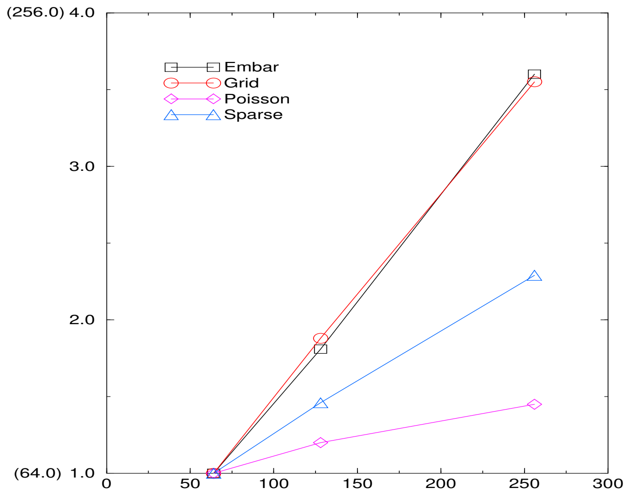

Figure 1: Benchmark Speedups for TMC CM-5

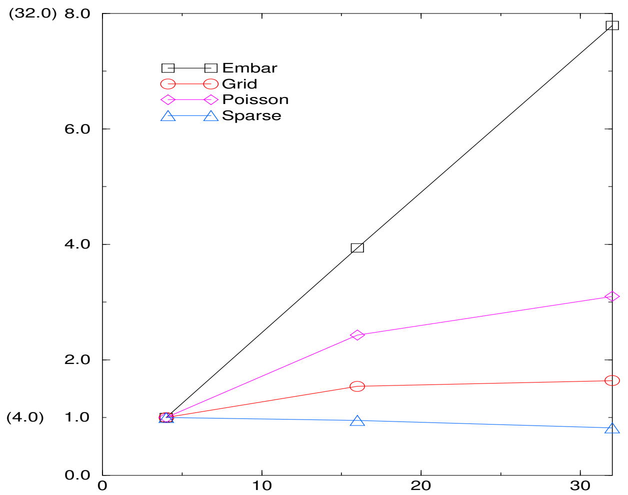

Figure 2: Benchmark Speedups for Intel Paragon

[21] D. A. Reed, R. A. Aydt, T. M. Madhyastha, Roger J. Noe, Keith A. Shields, B. W. Schwartz, *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, November 1992.

[22] Z. Segall, et al., *An Integrated Instrumentation Environment*, IEEE Transactions on Computers, Vol. 32, No. 1, pp. 4–14, Jan., 1993.

[23] Z. Segall and L. Rudolph, *PIE: A Programming and Instrumentation Environment for Parallel Processiong*, IEEE Software, Vol. 2, No. 6, pp. 22–37, Nov., 1985.

[24] V. S. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, Concurrency: Practice & Experience, Vol. 2, No. 4, pp. 315–339, December 1990.

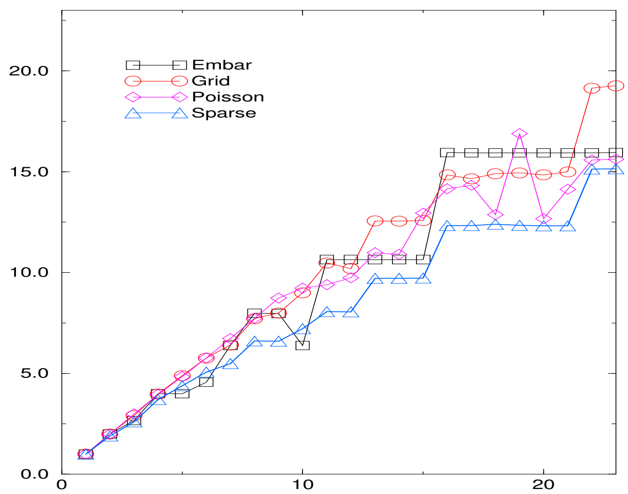

Figure 3: Benchmark Speedups for Sequent Symmetry

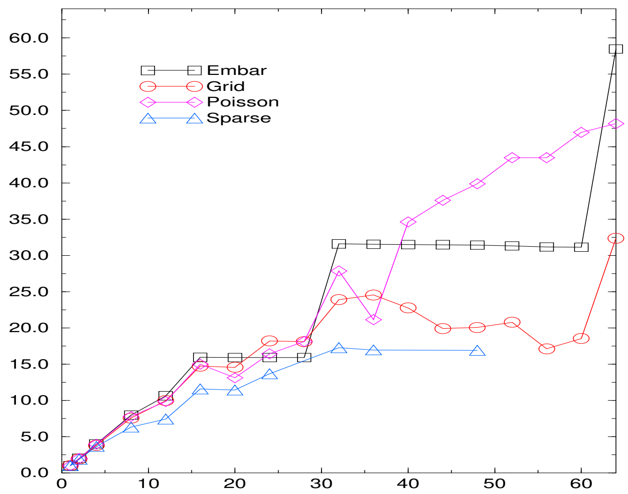

Figure 4: Benchmark Speedups for BBN TC2000

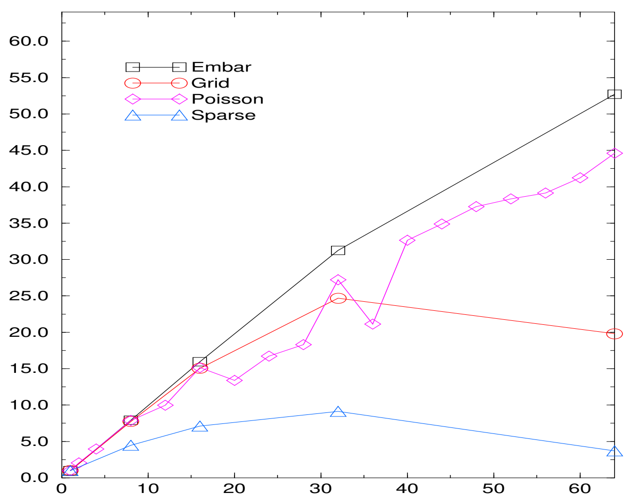

Figure 5: Benchmark Speedups for KSR KSR-1

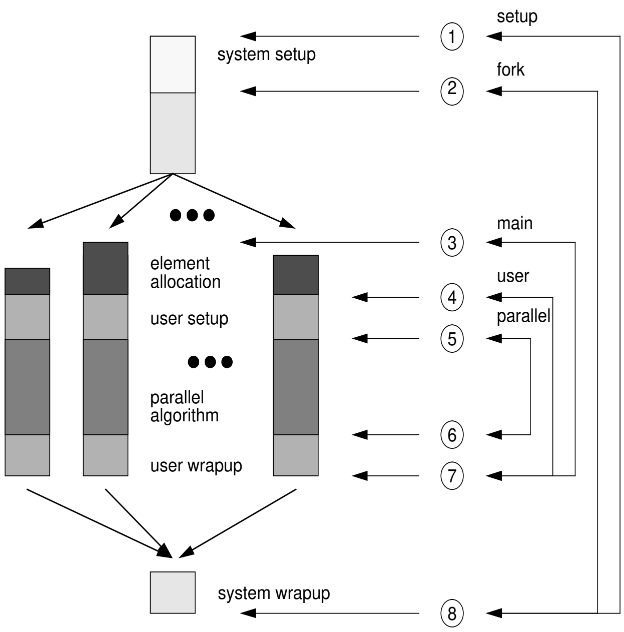

Figure 6: pC++ Execution Phases for Measurement

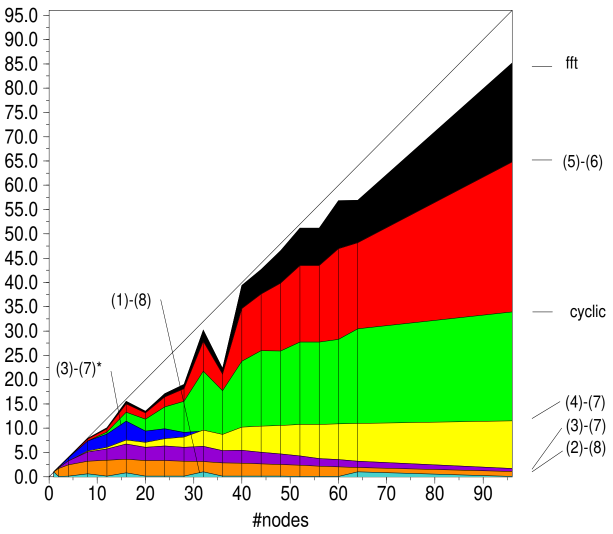

Figure 8: Detailed Speedup Profile: Poisson on BBN TC2000

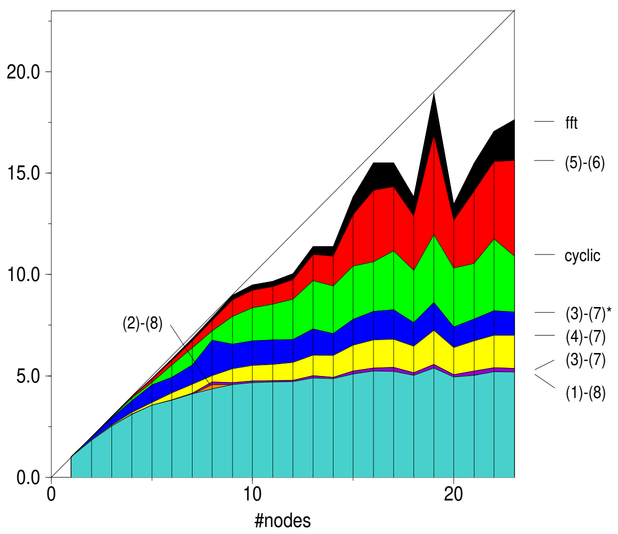

Figure 7: Detailed Speedup Profile: Poisson on Sequent Symmetry

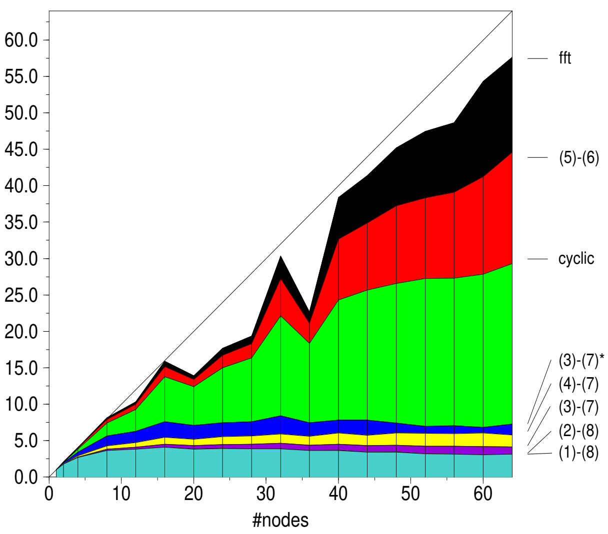

Figure 9: Detailed Speedup Profile: Poisson on KSR KSR-1