PARC SIMULATOR ON WINDOWS-NT

by

Ayal Itzkovitz

Supervised By
Dr. Assaf Schuster
# Table Of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TABLE OF CONTENTS</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>ABSTRACT</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>INTRODUCTION</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>THIS DOCUMENT</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>PROJECT PURPOSE</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>PARC PRECOMPILED</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>WINDOWS-NT</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>USING THE SIMULATOR</strong></td>
<td>7</td>
</tr>
<tr>
<td><strong>COMPILING A PARC PROGRAM</strong></td>
<td>7</td>
</tr>
<tr>
<td><strong>EXECUTING THE PARC PROGRAM ON TOP OF THE SIMPARC</strong></td>
<td>7</td>
</tr>
<tr>
<td><strong>DESIGN AND IMPLEMENTATION OVERVIEW</strong></td>
<td>9</td>
</tr>
<tr>
<td><strong>GENERAL</strong></td>
<td>9</td>
</tr>
<tr>
<td><strong>ACTIVITIES</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>SHARED MEMORY</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>EVENTS (MESSAGE PASSING)</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>INITIALIZATION</strong></td>
<td>11</td>
</tr>
<tr>
<td><strong>VP (VIRTUAL PROCESSOR)</strong></td>
<td>11</td>
</tr>
<tr>
<td><strong>LAM (Local Activities Manager)</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>The RAL</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>RALPtr</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>Activity Termination</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>SE (SIMULATOR ENGINE)</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>IMPLEMENTATION</strong></td>
<td>13</td>
</tr>
<tr>
<td><strong>THE MODULE: EVENTS</strong></td>
<td>13</td>
</tr>
<tr>
<td><strong>CONTEXT SWITCHING</strong></td>
<td>14</td>
</tr>
<tr>
<td><strong>The RAL</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>RALItem</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>The RALPtr</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>STATE NEW ACTIVITIES</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>TERMINATING ACTIVITIES</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>BLOCKING CONSTRUCTS</strong></td>
<td>18</td>
</tr>
<tr>
<td><strong>SuspendMyself() implementation</strong></td>
<td>19</td>
</tr>
<tr>
<td><strong>MUTUAL EXCLUSION</strong></td>
<td>19</td>
</tr>
<tr>
<td><strong>INSERTING AND REMOVING TO/FROM RAL</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>InsertToRAL</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>RemoveFromRAL</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>THE MODULE: SIMFUNC</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>switch_off / switch_on</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>Yield</strong></td>
<td>21</td>
</tr>
</tbody>
</table>
Abstract

ParC 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. The language ParC was developed in the Hebrew University of Jerusalem.

Although ParC is a parallel programming language that intend to be running on a multi-processors machine, we can run in on a single-processor machine using a simulator application, simulating a program execution on a multi-processors machine.

In this paper we cover the ParC simulator design and implementation. The simulator runs on any machine running Windows-NT operating system.

In the following section you will find a general introduction of various related topics to the ParC simulator (the language, the preprocessor and Windows-NT).

In the third section you will find a brief user manual on “how to use the simulator”. After reading this section you should be able to take the SimParC executables and run ParC programs upon the simulator.

In the fourth section you will find an overview of the simulator’s design. After reading this section you should be familiar with the main components of the simulator (SE, VP, events and initialization) and their relations.

In the fifth section you will find description of the implementation of the several modules, composing the simulator, including various topics, we considered as important like: starting and terminating activity, blocking constructs and the RAL management. The runtime implementation of the PCC generated function calls is explained in details. After reading this section you should be understanding the SimParC main components’ implementation, as well as the runtime implementation of the PCC generated calls.

The sixth section describes briefly the statistics, the simulator gains during execution. This part of the simulator is quite primitive and thus described in short. After reading this section you should understand the data you get when simulator is executed. Note that a more profound tool for profiling and tuning the parallel ParC programs was developed in the Technion’s Windows-NT laboratory, visualizing the parallel execution of a ParC program over the SimParC. This tool and other currently developed tools’ manuals should be acquired separately.

The seventh section describes briefly the main future extensions we are working on. For more updated details please contact via email: ayali@cs.technion.ac.il or assaf@cs.technion.ac.il.

The eighth section contains relevant bibliography. Some of the articles have not been released and can be found in URL: http://www.cs.technion.ac.il/~ayali/parc/.

The last section contains a chart of the project tree, including a one line description of the files in each directory. The sources are intended to be placed soon in the above WWW page.
Appendix A contains a brief ParC reference with syntax for most ParC constructs and commands. Appendix B contains a description of the Visualization tool that was developed to the ParC simulator.
Introduction

This Document
This document uses many of the ParC phrases and terms. The reader of this document should be fully familiar with the ParC language and its main conceptions.

Project Purpose
The ParC simulator for Windows-NT (we will call it from now on - “SimParC”) is designed to use the special features of Windows-NT to accomplish a portable simulator that can run on any machine running Windows-NT, independent of its hardware.

ParC Precompiler
ParC is a programming language and thus should have its own compiler. Because ParC is an extension to C, it can use the entire Ansi-C grammar rules with few additional rules for ParC constructs and keywords. The ParC language is beyond the scope of this document and should be acquired separately [4].
A ParC precompiler (we will call it “PCC”), converts ParC program to C program, while ParC constructs are converted to C function calls, which are implemented by the SimParC. Further information about PCC can be found in the PCC documentation [3].

Windows-NT
Windows-NT is an operating system, from Microsoft Corp. Its main features were in the core design of SimParC.
Windows-NT main features: Preemptive multitasking; Processes with virtual 32 bit address space; Threads as part of a process; Win32 subsystem with threads and processes handling 1APIs; Portability and Inter Processes Communication (IPC).

1API = Application Programming Interface
Using The Simulator

The SimParC simulates a multi-processors machine. It works in console mode, providing a different window for every Virtual Processor (which is a simulated processor), as well as a window for the Simulator Engine (which is the central unit of the simulator). Some other graphical windows are provided by other tools, connected to the simulator (like the SimParC visualization tool).

The simulator is easy to use and for the user’s point of view consists of two phases:

• Compiling a ParC program.
• Executing the ParC program on top of the SimParC.

Compiling a ParC program

A program written is ParC should be pre-compiled by PCC to a C program. Afterwards, it should be compiled and linked to a \(^2\)DLL, together with a special library that the SimParC supplies (SimFunc.Lib). This library resolves the SimParC function calls.

A special script was written for automating the above procedure, named: ParC2DLL. This script gets the ParC source file and builds a DLL named: Simulate.DLL.

Use the syntax:

\[ \text{parc2dll <parc src file name}> \]

When finished, a the generated DLL, named Simulate.DLL can be executed upon the simulator. Its name can be changed to any valid name (with .DLL extension).

Executing the ParC program on top of the SimParC

Two executables should reside in the path: \text{simparc.exe} and \text{procesor.exe}.

For execution use the following syntax:

\[ \text{SimParC <num of VirtualProcessors> <DLL name> } \]

Several windows are created: a window for each Virtual Processor (VP) and a window for the SE (Simulator Engine) itself. If other SimParC tools has been installed, some other windows can be opened as well, as appropriate to the activated tools.

When execution ends, the SE window contains the statistics for each simulated processor, accumulated during simulation. All VPs’ windows are closing automatically in the end.

\(^2\)DLL = Dynamic Link Library
Figure below shows the stages in preparing a ParC program to run:

ParC source code

PreCompiler (PCC)

C file of ParC program

SimFunc Library

Compile & Link

ParC program DLL

Run-Time Layer

VP executable

VP executable

VP executable

SE executable
Design and Implementation Overview

General
SimParC is a simulator for multi-processors environment, runs on a single machine. SimParC consists of a single process, named “Simulator Engine” (SE), and an additional process for each simulated processor, named “Virtual Processor” (VP). Activity is implemented by a thread, that runs on the appropriate process who represents its assigned processor (VP). All activities are created suspended and are being resumed and re-suspended by the VP. The SE, as its name implies, is the engine of the simulator. It assigns activities to processors, responsible for managing the activities tree (tree of parent-son relations) and the flow of the simulation processes. Any VP acts as a standalone processor, which gets activities to be run on, and performs context-switches between them. It doesn’t know about the relations between the activities (such as: parent of, brother of, etc...). On the other hand, the SE is the main heart of the simulator: it stores the activity tree, knows all relations and locations of each activity and responsible for the connections between them.

In ParC there are several constructs, which are strongly related to the activity’s parent and descendants (e.g. sync). This kind of constructs, transfer requests to the SE, notifying the activity’s family of reaching such a construct, and the SE passes it along to the appropriate activities. There are also constructs (like switch_off) which do not require the knowledge of the family relations and hence are implemented locally in the VP.

Below is a chart of the general structure (processes and connections) of the simulator:
Activities
Each activity is, in fact, a thread, running a precompiler generated f_p_a_r# function (except the main one that runs the function main()). All f_p_a_r# functions are exported in the dll that the simulator uses, and therefore can easily called using GetProcAddress() API. When f_p_a_r# finishes, the thread also finished running its procedure and after some cleanup (sending event to SE telling that he is going to die), it terminates normally.

Shared Memory
ParC assumes a shared-memory model. On Windows-NT each process has its own 32 bit address space and there are special tools to share memory between processes on same machine.
Upon initialization, each VP reserves a fixed size block that its data will be shared by all VPs. Because we cannot guarantee that all blocks are allocated in same virtual address, each VP saves its starting address of its shared block. All VPs shared blocks' virtual address spaces can be viewed by each other VP. We open views for all other VPs’ blocks by using File Mapping APIs (CreateFileMapping(), OpenViewOfFile(), etc...). Variables are passed between threads on different processes by their offsets is their shared block. We couldn’t simply pass pointer because of possible differences in the shared blocks’ starting addresses.
The Shared Memory Manager (SMM) is responsibly for allocations and deallocations from the shared memory block. Our SMM has its own data-structure that contain the free memory regions in a linked list. Allocations use first fit algorithm.
Exported functions to other modules:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share()</td>
<td>Allocate space from the shared block.</td>
</tr>
<tr>
<td>UnShare()</td>
<td>Deallocate a recently allocated space from the shared block.</td>
</tr>
<tr>
<td>LocalPtrToOffset()</td>
<td>Convert a pointer in the shared block of a VP to an offset in the share block.</td>
</tr>
<tr>
<td>LocalOffsetToPtr()</td>
<td>Convert an offset in the shared block to a pointer in the VP’s shared block.</td>
</tr>
<tr>
<td>RemotePtrToOffset()</td>
<td>Convert a pointer in other VP’s shared block to offset in the shared block.</td>
</tr>
<tr>
<td>RemoteOffsetToPtr()</td>
<td>Convert an offset in the shared block to a pointer in other VP’s shared memory block.</td>
</tr>
<tr>
<td>ParentOffsetToPtr()</td>
<td>Convert an offset in the parent activity’s shared block to pointer.</td>
</tr>
</tbody>
</table>

Events (Message Passing)
As we saw in previous sections there should be an information path between a VP and the SE and between the SE and each VP. In our simulator's implementation we used the Windows-NT recommended method for interprocess communication: Named Pipes.
Named Pipes supply a convenient way of interprocess communication including message-passing. By creating and opening a named pipe between two processes, we can use the handle we get as a file handle, performing reads and writes from that pipe as if it were a stream file. Also supplied an overlapped I/O support and event notification for each write operation on an empty pipe.

In Windows-NT, pipes are bi-directional. We opened a pair of pipes for each (SE,VP) pair, so that each pipe is used as a uni-directional pipe. Hence, each VP has two pipes: from one of both it only reads and to the second it only writes. We chose that way because of the events notification of Windows-NT, that does not identify read/write events as different events on same pipe.

Special purpose threads in both the VPs and the SE wait for the 3 outside-in pipes to get filled, and once new bytes are written (we call it: event occurred), it handles that data (which we call: event’s info). For all kind of events, event’s info is filled with an EVENT_CONTEXT structure, always written and read to/from pipe as a complete structure.

**Initialization**

When the SE recognized that enough VPs had connected to it, initialization starts. The table below shows the initialization process of SE and a VP.

<table>
<thead>
<tr>
<th>SE</th>
<th>VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create enough pipes and wait for all VPs to connect.</td>
<td>Try to open pipes to/from SE.</td>
</tr>
<tr>
<td>Assign and send each VP a unique ID.</td>
<td>Create a block of memory to be shared by all VPs. Send FINISHED_INIT to SE.</td>
</tr>
<tr>
<td>When got FINISHED_INIT from all VPs, send INIT_COMPLETE_ALL to all VPs</td>
<td>Load the simulated DLL. Initialize simulation statistics. Initialize RAL.</td>
</tr>
<tr>
<td>Send event to a VP to run the main activity.</td>
<td>Open views for all shared blocks of other VPs.</td>
</tr>
</tbody>
</table>

Initialization processes is in phases, where events are sent to SE from each VP when it finishes a phase and when the SE recognized that all VPs had finished their phase, it sends events for all VPs to advance to next phase. Finally, the SE chooses a VP and tells him to run the main activity.

**VP (Virtual Processor)**

Each activity, running on a VP, has a thread associated with it. Upon creation of that thread, we allocate a hash node VP global hash-table with the thread specific data which remains until thread’s termination. This data contains: thread details (id, handle, etc...), RAL item (to be discussed later) and list of shared allocations that the thread performed.

---

3A pipe which the current thread only reads from.
LAM (Local Activities Manager)
A VP can have many threads, each of those want to use the cpu. Windows-NT manages threads with same priority in round-robin policy, while quantum equals to 20ms. This fact causes us problem if we want different duration between context-switches. We also need greater control of when and how do context switches occur. Our solution is to maintain a high-level primitive context-switch: except the currently running thread in the VP, all other threads (activities) are usually suspended; when its timeslice finishes, it is being suspended and another suspended thread is being resumed. Windows-NT supply us the APIs: `SuspendThread()` and `ResumeThread()` for these purposes and we assign a special thread for scheduling named: LAM (Local Activity Manager).

The RAL
The Running Activity List is called RAL. Each thread has a hash node, associated with it, that contains a RAL item. The RAL item contains a "virtual context" of an activity together with activity related data, and is linked in a linked list. When the LAM reaches a decision that a context-switch should be made, the current activity is suspended, new activity is chosen from the RAL (using the links), its "virtual context" is logically being taken and restored on VP (by only advancing a pointer) and thread is being resumed. This way we prove round-robin between all activities as well as simple control of which activities are enabled or disabled to run on VP.

RALPtr
RalPtr is a VP-global pointer that points to the current running activity’s RAL item. RalPtr is valid (that is: correct) when wrapped by the pair of ParC commands `switch_off/switch_on`. Proof of correctness later.

Activity Termination
When activity terminates, naturally or unnaturally, its allocated shared data is freed. All allocated shared data exists in a linked list in the hash node (next field after the activity’s RalItem) and thus deallocating all data is easy to determine. Thread's RAL item is being unlinked from the RAL and its hash node is being freed. If the thread, needed to die was running, a context-switch had been made before.

SE (Simulator Engine)
The SE does nothing but serving events, arriving from all VPs. SE is designed as a single thread, having open pipes from/to every VP and its main loop is for waiting on all outside-in pipes for an event, and once it arrives, serve that event, usually by marking activity’s states and post other events to related VPs. SE maintains the activities’ tree (i.e. collection of all activities’ threads: location, state, parent of and sons of relations). In the common execution, the SE handles `sync` requests, sending rerun events to all VPs when sync is satisfied. Whenever an activity executes a parallel branching (`pparblock` / `pparfor`), the SE is responsible for choosing processors on which to run new activities, and also to post the restart events. Load balancing algorithms can be implemented in the SE process.

In the implementation section we will discuss how the SE is involved in performing ParC commands.
Implementation

The Module: Events
SimParC uses its events to pass information between two distinct processes (e.g. the SE and a VP). The process of posting an event:
1. Define a variable of the type: EVENT_CONTEXT. Assume ec.
2. Fill ec with event’s type.
3. Fill ec with the event’s information.
4. Choose the pipe you want to send the event through. This pipe should be an inside-out pipe. Assume it is OnPipe.
5. Post event via SEPostEvent(OnPipe, ec); or VPPostEvent(OnPipe, ec);

The SE and the VP use SEPostEvent() and VPPostEvent() respectively. Pipe operations are stream like. When opening a pipe, we attach an event to be signaled whenever new data is being written or read to/from it. Because we use each pipe as a uni-directional pipe, we check only the events on the side that reads the events, which gets the event signaled each time a new event is being written. Hence, the algorithm for the writer is:
1. Write to the pipe.
2. Wait until the reader read that event.
and for the reader:
1. Wait for the event of the outside-in pipe to be signaled.
2. When signaled, read the event’s information.

A problem arises when the couple of processes in both sides of the pipe want to post events (on different pipes, of course). A deadlock might occur in the following scenario:

---

4 real Windows-NT event
As we see P1 waits to P2 to read the event's info from pipe, and until it doesn't, P1 won't get freed to read P2's event info. Same occurs in P2, and therefore both are waiting for each other to read the event's data and only afterwards, each of them will read its incoming data.

The solution was to break "the circle" below, to couple of threads: a thread for reading an event as it arrives, and a thread for posting the event on the pipe. Only the SE implementation changed (because always it is a side in the circle), so that now each event, the SE wants to post, is stored in a queue, (named: EQueue) and a semaphore synchronize the sender thread, who posts the events on the pipe.

Algorithm for both threads in the SE are:

For the sender thread:

1. While semaphore value is positive (not zero), post event by writing to the pipe and waiting for the other side to read it.
2. If semaphore's value became zero (or negative), wait on the semaphore.

The receiver thread hasn't been changed.

SEPostEvent() implementation is now:

1. EQueue.AddNewEvent().
2. Signal the EQueue's semaphore.

Chart below shows the event mechanism:

---

**Context Switching**

In order to have a full control on the running activities on each processor we provide a higher level task switching mechanism than the operating system provides. In this article we use the term "context switch" for this higher level context switch, explained below.
Each VP has its LAM (Local Activity Manager), which acts as the processor’s operating system’s scheduler. We use the Win32 APIs SuspendThread and ResumeThread for allowing and disallowing a thread to run. Each LAM runs in the main thread of each VP process and uses the policy of enabling only a single thread to run simultaneously. A simulator defined timeout is declared to simulate a quantum mechanism on a multi-tasking environment. Each time this timeout expires, a context switch performs and a new thread is chosen to be run on the VP.

**The RAL**

The Running Activities List (RAL), is a linked list, built over the VP’s main threads’ database, that holds the currently running activities in the VP. The figure below shows the general structure:

![RAL Diagram](image)

**RALItem**

Each item in the RAL list is called: RALItem. A RALItem contain various thread specific information which is the thread's ParC-context. Below is a complete listing of the fields in the RALItem:

- **ActID**: Activity ID of current activity.
- **ThreadHandle**: Thread’s handle.
- **ContextSwitchCounter**: A cumulative counter for enabling/disabling context-switching while this thread runs on its processor.
- **BlockCount**: Used for blocking calls.
- **hUnBlockEvent**: Used for blocking calls.
- **hYield**: Handle to be used by yield().
- **next**: next RALItem in the RAL.

**The RALPtr**

The RALPtr is a pointer to the RAL item in the RAL, corresponding the currently running thread.
An important invariant in the SimParC implementation is:

"Between a switch_off() / switch_on() pair, the RALPtr points to the RALItem of the currently running thread".

The importance of this invariant is significant. Each query of the thread’s maintenance data (activity’s data) which is stored in the RALItem, is promised to be valid and point to the correct ParC context if used between switch_off/switch_on.

Stating New Activities
Activities are created as a result of ParC’s parallel constructs (e.g. pparblock / pparfor). The operations, being taken stage by stage by the various SimParC components, are described below:

First, the thread, who called the parallel construct, allocates an array of function calls. Each element in the array is related to a different son activity’s starting position. In the elements of the array, the thread inserts the function name, number of parameters to that function, the origin processor (that is the current processor) and a pointer to another array, allocated in the shared memory which contains the parameters for the function in a double-word alignment (4 bytes).

After those preparations, an event is being posted to the SE, asking him to create new son activities. Along with the event the above pointer to the function call array is sent. After sending that event, the thread is suspended, and would be resumed when the SE recognizes that all his sons are dead.

When The SE gets that event (ACT_CREATE), it creates new son nodes in the appropriate place in the activities tree, chooses processors for the son activities (currently: round-robin between all processors), chooses an activity id, and for each activity, posts event (ACT_CREATE) to its newly assigned processor, indicating the pointer to the array of function calls in the parent’s shared memory block, and its entry in that array.

The creation of the activity’s thread is actually done in the third stage, when the ACT_CREATE event is being served by the target process.

Because a new thread’s starting routine can be given only a single parameter, all parameters to the new thread are packed in a structure (SimActWrapperParam) and a pointer to that structure is provided as the thread’s starting routine’s parameter. The routine, which the new thread will run, is a special purpose routine (named: SimActWrapper) which wraps the call for the user “f_p_a_r#” routine. This routine resolves the function’s address by calling the API FindProcAddress(), finds the appropriate parameters’ array and calls the preprocessor’s generated “f_p_a_r#” routine with it. In this stage, a new thread is created in suspended state using CreateThread() API, running the SimActWrapper routine. The new thread is assigned a structure, where several details are kept including its ParC’s context (its RALItem).
Such structure looks as follows:

```
ThreadID
  RALItem
    Share List
      next

ThreadHandle
ActID
ContextSwitch Counter
BlockCount
hUnBlockEvent
hYield
ParentProcessor
```

The last thing, left to do, is to insert the thread’s RALItem into the RAL, so the LAM can context switched with it.

**Terminating Activities**

Activity can be terminated in two ways: Naturally and Unnaturally. Naturally - when it has completed executing its code. Unnaturally - when other activity has performed `pbreak` or `preturn` which leaded to its termination while it was running.

Due to the parallel behavior of the termination, dividing implementation for natural and unnatural termination can lead to a very complicated correctness proof, while handling both cases in a single comprehensive algorithm can be the optimal solution.

Catching all termination of activities is vital. When activity terminates it should leave the RAL, free its internal data-structures in the processor and the SE and free all shared variables, it has allocated. Because son activity might use its parent’s allocated data, termination must perform Bottom-UP.

To accomplish a Bottom-Up termination we implemented termination in two stages: First, freezing all activities in the subtree we want to terminate. Second, for each frozen activity, when all activities in its subtree are known to be frozen, activities are deleted and their data-structures and shared variables are being freed. Notice that freeing is done bottom-up immediately when possible without waiting for first stage to totally complete.

When decided to terminate an activity, SE performs the following for each activity in its subtree beginning Bottom-Up:

1. Decrease parent’s number of living sons.
2. Increase parent’s number of frozen sons.
3. If the activity to be frozen is waiting for sync, decrease parent’s counter for synced sons.
4. Mark activity as frozen.
5. Post event to activity’s processor to freeze activity.

Processor, upon receiving the freeze event does the following:

1. Remove activity from RAL (even if it is running now).
2. If trying to free current running activity, perform context switch.
3. Post event to SE, notifying that activity is frozen.
Activity’s RALItem is being removed from RAL and thus won’t run again in future. We also force context switch if (and only if) the activity is running when getting the freeze request, so that when notifying the SE that the activity is frozen, it can be sure that it’s internal structures and shared variables can be freed.

The SE, when getting the freeze notification does the following:
1. If activity has frozen sons, do nothing and exit.
2. Mark activity as deleted.
3. Unlink current activity’s node from the activity tree.
4. Post event to appropriate processor to free activity’s structures.
5. If activity’s parent has no sons anymore act according to its state:
   - if deleted, perform this procedure to it.
   - if frozen, do nothing.
   - if running, resume parent by posting event to its processor.

If SE gets the notification of frozen event, it is promised that this activity doesn’t and won’t run anymore. If activity has frozen sons, it means that we didn’t get notification from all its descendants that they are frozen, and therefore we mustn’t free activity’s structures (including freeing its allocated variables). In this case, we go out. When activity gets the notification for being frozen and all of its sons are deleted or marked as deleted, it is promised that its structures and allocations can be freed, and indeed, an event, asking for freeing the activity’s structures is sent to appropriate processor.

Because activity can marked as deleted but actually stay frozen (e.g. in the case when frozen notification has accepted for the activity but its son hasn’t got its notification yet), the parent activity’s state is also checked and if it is marked deleted, all procedure applies to it again. If parent activity is in running state (and hence should be resumed when all its sons are dead), it is checked if it can be resumed and if so, an appropriate event is being posted to its processor. Notice that this it the mechanism for resuming a suspended activities due to son activities creation.

**Blocking Constructs**

Each ParC activity is implemented as a Windows-NT thread. The following ParC constructs are blocking constructs: pparblock, pparfor, sync, pcontinue, pbreak and preturn.

For correct execution of ParC program we implemented the blocking mechanism as explained below:

Each blocking function includes a call to SuspendMyself() before it ends. SuspendMyself() guarantees that the current thread is blocked (i.e. next instruction doesn’t execute) until an explicit resume request arrives.
**SuspendMyself() implementation**

1. `switch_off`: disallow context switching
2. Signal a processor global event, notifying the LAM that current thread wants to suspend itself.
3. Wait until the processor’s global event for acknowledgment on received suspend notification is signaled.
4. Take a thread specific event for unblocking (taken from RALItem of current thread).
5. `switch_on`: restore previous context-switch state.
6. If context switching is disabled: yield
7. Wait for the unblock event to be signaled.

When LAM sees that the suspend myself request event is signaled it does the following:

1. Decrease the BlockCount of the requesting thread.
2. If block count became zero: signal the thread’s specific unblock event. else: remove thread from RAL.
3. Signal the acknowledgment event for suspend request received.

When resume request arrives to the LAM, it performs:

1. Increase thread’s specific BlockCount (taken from its RALItem).
2. If BlockCount became non-zero (equals 1), exit.
3. Insert resumed thread to RAL
4. Signal thread’s specific unblock event.

Usually, threads post an event to the SE and afterwards perform SuspendMyself(). The resume request can be served before the SuspendMyself completes and thus a special care should be taken in implementation of blocking operations. We used a counter and an handshake mechanism for implementing the blocking operations. First handshake, using two events (for request and for acknowledge), assures that the request is served by the LAM, mainly by removing the activity from the RAL. When wait on acknowledge event is satisfied, thread can wait on the unblock event, which is a thread-specific event, created when activity’s thread first initialized. The unblock event will let the thread continue its execution. When resume arrives at the LAM it has to signal the thread’s unblock event and insert it again to the RAL. When context switch occurs, thread can continue running.

A problem arises if a resume request arrives before SuspendMyself ends. Solution consists of a counter, decreased by the LAM when removing blocked activity from RAL (serving the SuspendMyself request event), and increased by the LAM when got a resume request. Only when this counter becomes zero, the unblock event is signaled by the LAM. That assures that if resume arrives before suspend completes, the counter will become 1, and when SuspendMyself is served, the activity won’t be removed from RAL; Instead, it will reset the counter and release the activity by signaling its unblock event.

**Mutual Exclusion**

Only few code fragments should have closed in critical sections. A single variable has been declared to accomplish minimal mutual exclusion in the processor’s code. Only the context switch main function (PerformContextSwitch()), `switch_off` and
switch_on are mutually exclusive, using the above variable. Note that this was enough to assure that the main invariant about RALPtr validity is correct.

**Inserting and removing to/from RAL**

**InsertToRAL**
Link current RALItem to the linked list.

**RemoveFromRAL**
Basically: unlink from list.
RALPtr should have a special care. Because removal from the RAL is only logical unlink from a list, RALPtr can point to a RALItem that is not in the linked list. Example of such scenario below:

![Diagram of RAL items and RALPtr connections](image)

Notice that even if RALPtr points to an unlinked RALItem, context switch will act correct, choosing the next RALItem, simply by choosing RALPtr→next.
When removing from RAL, we also check if we remove the RALItem pointed by RALPtr→next. If so, we must update RALPtr→next to point to the RALPtr→next→next.

**The Module: SimFunc**

**switch_off / switch_on**
Disable and enable context switches on processor.
Implementation has a semi-commutative behavior. Switch_off’s and switch_on’s can be performed in a pair inside a pair manner. A counter, in the RALItem of the current running thread (RALPtr->ContextSwitchCounter), accumulates the number of switches, while switch_on decreases the counter and switch_off increases it. The current state of the processor can be determined by looking at the counter, the RALPtr is pointing to, and if it is less than zero, context switches are disabled. Many switch_offs are accumulated and same number of switch_ons must perform to enable context switches again but on the hand no accumulation perform for enables, which means, each switch_off promise to make the counter less than zero (context switches disabled) so that a programmer can use switch_off for safely disable context switching in the processor.

An important thing is that determining whether context switch is enabled or disabled currently on the processor is being done by examining the RALPtr, which gives us the context behavior of the activity's data. Whenever a thread replaces other, it is automatically "loading" its context by only advancing the RALPtr to the next RALItem in list.

switch_off and switch_on are part of ParC’s reserved words and can be used by the ParC programmer, but they are also used internally in the simulator’s implementation to promise the correctness of the RALPtr. Recall the invariant before, correctness of any RALPtr usage between switch_off/switch_on is promised. The reason for this is, informally, because the switch_off and switch_on are mutually exclusive with and hence when performing switch_off, it is promised that context-switch code is not running and won’t run until switch_off hasn’t finished. While finished, context switches won’t perform until switch_on is called. The only exception for this is whether yield is performed, which force context switching, and thus can harm the correctness. Therefore, yield implementation is a bit complicated. It is described later.

Yield
The naive implementation would probably be a single call to the context switch routine, in order to switch tasks. Due to the fact that a powerful invariant can solve many correctness problems and decrease the complicity of the implementation, the decision was to overcome the yield implementation in a way that won’t weaken the invariant.

An activity’s thread, along with all other components, is also assigned an event, named hYield. This event is being used to notify the thread that his yield request has been satisfied and completed. The handle of the event is being taken from the RALPtr between switch_off/switch_on and being held in a local variable. A yield request is being raised (implemented as a common event for all threads for asking yield from LAM), and a wait is performed on the taken yield event. Once the LAM serves the yield request (which has a higher priority than other SE events), it forces context switch and only afterwards sets the thread’s yield event. When this thread would be context switched to work again, it would have the yield event signaled and simply continue its work, knowing that his most recent yield request has been fulfilled. Note that we preserve the invariant that between switch_off/switch_on pair, the RALPtr’s correctness is promised.

pparblock / pparfor
See “Starting New Activities” above for detailed description.
**pcontinue**
A brief definition might claim: The construct "pcontinue" is for a parallel construct like "continue" is for a "while" construct in the C language. Whenever pcontinue is reached inside a parallel construct, the current activity terminates. Thus, implementation of pcontinue is:
1. Post event to SE telling him that current activity performed pcontinue.
2. SuspendMyself.
Implementation is very simple and accurate: current activity is being blocked after posting the event to SE. SE, upon getting that event, behaves as if it got an activity termination event and it handles it according (see "Terminating Activities" above).

**pbreak**
As we wrote for continue, pbreak might be briefly defined as: "pbreak" is for a parallel construct like "break" is for a "while" construct in the C language. Whenever pbreak is reached inside a parallel construct, the whole subtree, which its root is the activity's parent, is terminated. Thus, implementation of pbreak is:
1. Post event to SE telling him that current activity performed pbreak.
2. SuspendMyself.
Like in pcontinue, implementation is simple: current activity is being blocked after posting the event to SE. SE, upon getting that event, terminates the whole appropriate subtree (the one that its root is the activity's parent). Activity's parent is automatically resumed when the last activity that needed to be terminated notifies that it has indeed terminated. For more details about termination see "Terminating Activities" above.

**preturn**
The construct preturn is performs "return" from the current user function, even if it is inside a parallel block. The "preturn" construct is strongly related to source file structure and hence it is fully implemented by ParC precompiler. The ParC precompiler mainly replaces the "preturn" calls with "pbreak" calls and therefore it is quite transparent to the simulator.

**sync**
sync is a major tool, a ParC programmer can use, for activity synchronization. "sync" can be viewed as a rendezvous between all brothers of an activity. Activity which performs sync is blocked until all its living brothers performed sync too. We call "sync satisfaction" the state when all living brothers of an activity are blocked due to a sync and thus should be resumed. We relate sync satisfaction to the parent of the activities who performed sync, and exists for this parent, all living sons are blocked by sync.
sync can be satisfied only in two cases: when activity performs sync or when activity terminates. Each when sync or terminate is performed, SE gets a notification event. It then updates the appropriate counter for the parent: number_of_synced_sons or number_of_living_sons, and if both counter have same value, it decides of sync satisfaction and release all sons. By defining the two occasions, where sync can be satisfied, correctness is assured.
Statistics

Overview
The simulator simulates a run of a parallel program on a multi-processors machine, using a single processor machine with a multi-threaded operating system. Statistics must be gained and reported in order to understand the simulation. It can be viewed as the "results" of the simulation.

The statistics we apply are currently:
1. Number of context switches on each processor.
2. Total time of processor executing in User-Mode.
3. Total time of processor executing in Kernel-Mode.
4. Idle time of each processor.

Number of context switches are counted every time a context switch is performed. Total time in user and kernel mode is accumulated whenever a thread terminates: it's times are queried and added to global variables. Idle time is taken from the NULL thread’s times (see The Null Thread below).

The NULL Thread
In order to know the time a processor was idle, an additional thread, called the NULL thread, is added. Whenever the RAL becomes empty, the NULL thread is inserted and start executing. When an activity arrives, the NULL thread is removed from the RAL.

When execution of ParC program ends, the running times of the NULL thread are transferred to the SE, as the total idle time of the processor.
Future Extensions

ParC is a language for writing concurrent programs on a shared memory environment. ParC programmer is encouraged to massively use shared variables and complex shared data-structures. A speedup can be achieved if ParC could run on a cluster of computers, running ParC services, sharing data among each other using distributed shared memory (DSM). We are currently examining the possibility to develop a distributed ParC environment, where a ParC program use cluster of computers running Windows-NT, and hope to achieve good speedups. Several released memory consistency algorithms would be checked using the distributed system.
Bibliography

[1] "Inside Windows NT" / Helen Custer
[3] PCC documentation / Ophir Holder & Nedim Frisko
[4] "ParC - An Extension of C for Shared Memory Parallel Processing" / Yosi Ben-Asher, Dror G. Feitelson & Larry Rudolph
Files and Directories

Following is a chart of directories structure is the project tree:

```
parc\src
  \inc
    Headers of the parc project
  \events
    Handling parc events between SE and VP
  \sharemem
    Shared-Memory implementation
  \simfunc
    Simulator's interface for the parc precompiled 'C' file
  \simparc
    The SE code (simparc.exe)
  \procesor
    The VP code: (procesor.exe)
```
Appendix A - Brief Reference of the ParC Language

The ParC programming language is a natural extension of the C programming language, providing parallel constructors and synchronization commands for maintaining concurrency.

<table>
<thead>
<tr>
<th><strong>pparblock</strong></th>
<th>Creates concurrent activities, running each a different block of code. Parent activity is blocked until all sons have finished execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pparblock {</td>
<td>block A</td>
</tr>
<tr>
<td>} : {</td>
<td>block B</td>
</tr>
<tr>
<td>} epar</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>pparfor</strong></th>
<th>Like sequential for, with all iterations executing concurrently. Each iteration is a standalone activity with a duplicated copy of var</th>
</tr>
</thead>
<tbody>
<tr>
<td>pparfor var;</td>
<td>start_val; end_val; step_val {</td>
</tr>
<tr>
<td>} epar</td>
<td>block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>sync</strong></th>
<th>Syncronize between brother activities. sync will block the activity until all its living brothers are also blocked in sync.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>pcontinue</strong></th>
<th>Ends current activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcontinue</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>pbreak</strong></th>
<th>Ends current activity and terminates all its brothers</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbreak</td>
<td></td>
</tr>
</tbody>
</table>
### preturn

| preturn(val) | Returns from current function, returning val as a return value. |

### switch_off

| switch_off | Disable context-switching on processor. |

### switch_on

| switch_on | Restore previous context-switching state on processor (the state before last switch_off). |

### yield

| yield | Force a context switch on processor (even if in disabled mode). |
Appendix B: Visualization Utility

Introduction
Graphical visualization aids to human comprehension of complex and large volumes of data. The behavior of parallel program is extremely complex and monitoring the performance of such programs generates vast quantities of data. So it seems natural that we want to use visualization in order to understand better program behavior and improve its performance. We run our performance meter on ParC simulator for Windows NT. Simulator consists of the set of virtual processors (each one is a separate NT process) and another process -- Simulator Engine. Each virtual processor simulates behavior of real processor in multi processor system on single processor hardware. The Simulator Engine initializes the simulation, distributes activities upon processors and maintains communication between virtual processors. By visualizing performance data we want to give to ParC programmer ability to understand easier and faster load on virtual processors, assuming that it will give him idea how work is distributed among processors and how he can improve the program to achieve maximal speedup.

Mechanism
We divide all virtual processor time into three parts: busy time, idle and overhead time. Processor is in busy state when it runs ParC activities. It is in idle state when it runs null thread (null thread is in running list only when there is no another ParC activity to run). Processor is in overhead state when it runs scheduler. We assume that processor and system load could be introduced as a distribution of these three periods. We define the time slice during which we will measure timing on each virtual processor. The presentation process looks like this: each given time slices the program measures processor timings and presents them on the screen in different ways. On our diagrams busy time has green color, overhead is yellow and idle is red (traffic light model).
Presentation

We have four different ways to present processor information:

- Utilization meter
- Processor count
- Gnatt chart
- Kiviat diagram
Utilization meter presents time distribution on each processor. It has a number of horizontal gauges equal to number of virtual processors in the system. Each gauge shows what time in percents corresponding processor was in different states on last measurement. The gauge is colored in three given colors proportionally with the time that processor was in each state. Also, each gauge has two numbers: left shows percentage of the time that processor was in *busy* state and right -- in *overhead* state. If gauge has white color only - it says that no measurement was made in last time period because of system overloading.

*Processor count*
Processor count diagram shows the time in percents that all system (number of virtual processors) was in one of three states -- busy, idle or overhead -- as a function of time. The timing of all system is calculated as a sum of corresponding processor time periods. Vertical axis presents 100% of time that all system acts in given time slice and horizontal -- presents the real time axis. The picture scrolls over horizontal axis according to progress of simulation process.

We draw current system state in following way:

At first we calculate percentage of time of each state in last time slice. After that, assuming that all height is 100% we calculate heights of bars that will represent different states and then draw each bar with corresponding color. States are drawn in following order from up to down: busy, overhead and idle.
This diagram shows the time distribution of each processor separately. It is presented with a horizontal bar chart (for each processor - another one) in which the color of each small bar indicates the state of the corresponding processor as a function of time. We determine processor state one time for each time slice according the follow simple rule: the state in which processor was the most of the time is called its state in this time period. Processor number is on the vertical axis and real time --- on horizontal axis, which scrolls according to simulation progress.
In this diagram each processor is represented as a stroke on a wheel. The recent average fractional usage of each processor determines two points on the stroke -- one for busy state and one -- for overhead. End of each stroke represents idle state point. Taken together the points determine three polygons whose sizes and shapes illustrate processor use and load balance. Inner polygon represents the distribution of busy time among processors and outer -- distribution of the idle time. Low usage concentrates the busy polygon near the center and poor load balance causes polygons to be asymmetric.

**User interface**
The performance meter is Windows program that uses MS-Windows GUI. It has push-down menus that allow user to open and close diagrams, to order windows and to change some visualization parameters. User could change length of time slice during which program updates internal representation of observed system. And also user could change update frequency which diagram uses to draw itself.

**Implementation**
Communication between performance meter and observed system is provided by Windows-NT Registry and Performance API calls. Performance meter starts before the observed system launches and writes its data to Registry. ParC Simulator reads the data and writes to registry necessary parameters that will give to performance meter ability to observe Simulator timings using API calls `GetThreadTimes()` and `GetProcessTimes()`.