Projektarbeit 1994/95

Projektdokumentation

Thema: Realisierung eines Batchmanagementsystems für den Parallelrechner

Jens Trützschler
01FIF91

Chemnitz, den 23. Juni 1995

Inhaltsverzeichnis

1 Aufgabenstellung **4**
1.1 Spezifikation der Aufgabenstellung 4

2 Untersuchung des Batchsystems DQS 3.x auf seine Eignung für die Belange des Parallelrechners **6**
2.1 Beschreibung und Funktionsweise des GC Powerplus 128 unter PARIX 6
2.2 Aufbau und Funktionsweise von DQS 8
2.2.1 Eigenschaften von DQS 8
2.2.2 Queue-Komplexe 8
2.2.3 Benutzung von DQS 8
2.2.4 Gesamtfunktion von DQS anhand von *qsub* 10

3 Installation des DQS-Systems, Realisierung notwendiger Anpassungen und praktische Erprobung einer Minimalvariante **12**
3.1 Installation 12
3.1.1 Systemanforderungen 12
3.1.2 Übersetzung und Konfiguration des Systems 12
3.1.3 Die Warteschlangen und Ressourcen des GC 15

4 Vergleich und Diskussion mit dem „Paderborner System“ Diskussion der Anforderungen aus URZ-Sicht **17**
4.1 Das Projekt Computer Center Software der Uni Paderborn . . . 17
4.2 Vergleich der Systeme 17

5 Ausbau der auf DQS basierenden Technologie **18**
5.1 Einleitung - ein DQS-Beispieljob 18
5.2 partd - Dämon zur Partition- und Jobverwaltung 19
5.2.1 Funktionsweise von partd 19
5.2.2 verwendetete Dateien und ihr Inhalt 20
5.2.3 die verwendete Klassenhierarchie 20
5.2.4 funktionelle Anforderungen an die Klassen 21
5.2.5 Scheduling-Strategien 22
5.2.6 Konfigurationsdaten 23
5.3 Das Informationsprogramm für die Jobs - inform_partd 24
5.4 notwendige Erweiterung von DQS 25
5.4.1 Analyse der Funktion von DQS 27
5.4.2 Die Kommunikation zwischen DQS 27
5.4.3 Modifikation von qconf 28
5.4.4 Implementation von qrerun 30
5.5 Installation des Dämonen 32

6 Integration der DQS-Technologie des Parallelrechners in die URZ-Technologie **33**
6.1 DQS-Systemadministration 33
6.2 Sicherheitsaspekte 33

6.3 Ausblicke zu DQS und partd 34

A Header-Files des Dämonen partd **36**
A.1 wichtige Voreinstellungen - defs.h 36
A.2 Klasse array - array.h 38
A.3 Klassen parttable, attribtable, sizetable - tables.h 39
A.4 Klasse allocsize - allocsize.h 41
A.5 Klasse complex - complex.h 42
A.6 Klasse config - config.h 43
A.7 Klasse daemon - daemon.h 44

B Warteschlangendefinitionen **46**
B.1 Queue gc_physic 46
B.2 Queue gc_attribute 47
B.3 Queue gc_128 48

1 Aufgabenstellung

1. Untersuchung des Batchsystems DQS 3.x auf seine Eignung für die Belange des Parallelrechners[1]
2. Installation des DQS-Systems, Realisierung notwendiger Anpassungen und praktische Erprobung einer Minimalvariante
3. Vergleich und Diskussion mit dem „Paderborner System“[2] Diskussion der Anforderungen aus URZ-Sicht
4. Ausbau der auf DQS basierenden Technologie für den Parallelrechner
5. Integration der DQS-Technologie des Parallelrechners in die URZ-Technologie

1.1 Spezifikation der Aufgabenstellung

Durch das URZ wurden folgende spezielle Anforderungen gestellt:

1. es sollen die freien Partitionen[3] verwaltet werden
2. neben DQS soll eine gleichzeitig interaktive Arbeit möglich sein
3. DQS soll dynamich konfigurierbar sein, d.h.
   - zeitgesteuerte Angabe des maximalen Anteils an Partitionen, die durch DQS angefordert werden dürfen
   - diese Zuordnung darf nicht durch feste Partitionenzuordnung[4] vorgenommen werden
4. DQS muß prüfen, ob ein Job wirklich gestartet wurde, im Fehlerfall muß dieser wieder, möglichst an erster Stelle, in die entsprechende Warteschlange aufgenommen werden
5. der Nutzer muß seinen Jobs folgende Attribute übergeben können
   - Partitionsgröße oder -bezeichnung
   - Anforderung eines speziellen Frontendrechners (ggf. mit oberem kombiniert)
   - Größenordnung der Laufzeit
   - Anfangspriorität des Jobs
6. aufgrund dieser Attribute sollen die Jobs in die enstprechenden Warteschlangen eingereiht werden unter Beachtung folgender Regeln

[1] wird allgemein als Bezeichnung des Parsytec GC Powerplus 128 verwendet
[2] gemeint ist hierbei das CCS - Projekt der Universität Paderborn
[3] als Partitionen werden die allokierbaren Einheiten des Parallelrechners bezeichnet
[4] z.B.: die rechte Hälfte des GC

- während der normalen Arbeitszeit[5] vorrangig Jobs mit kleiner Partitionsgröße und kurzer Laufzeit
- ansonsten Jobs mit großen Partitionen und langer Laufzeit[6]
- unabhängig davon sind Jobs mit hoher Priorität entsprechend zu bevorteilen
- Vor allem ist dafür zu sorgen, daß jeder Job in endlicher Zeit gestartet wird!

7. DQS soll eine Abrechnung der Jobs ermöglichen
8. bei grober Nichteinhaltung der angegebenen Joblaufzeit soll der entsprechende DQS-Administrator informiert werden.

[5]voraussichtlich in der Zeit zwischen 6 und 18 Uhr
[6]mehr als 64 Prozessoren

2 Untersuchung des Batchsystems DQS 3.x auf seine Eignung für die Belange des Parallelrechners

2.1 Beschreibung und Funktionsweise des GC Powerplus 128 unter PARIX

Der Parallelrechner der Firma Parsytec verfügt über:

- 128 Prozessoren MPC 601 („PowerChip“) in 64 Knoten (80 MHz Taktfrequenz, 80 MFlops peak performance double precision
- 64 * 4 T805-30 Kommunikationsprozessoren 35,2 MB/s Kommunikationsbandbreite pro Knoten, 8.8 MB/s Kommunikationsbandbreite zwischen benachbarten Knoten
- 64 * 32 MB Hauptspeicher
- Peak Performance des Gesamtsystems: 10 GFlops
- Gesamtspeicher: 2 GB

Der Zugang zum eigentlichen Parallelrechner erfolgt über zwei Frontendrechner (Sun Sparc10) mit den Namen **kain** und **abel**. Das Betriebssystem des Parallelrechners ist PARIX[1][2]. Diese Kurzbeschreibung wurde größtenteils aus [3] übernommen.

Zustandsabfragen am Parallelrechner

Um mit dem Parallelrechner arbeiten zu können, ist es notwendig bestimmte aktuelle Parameter abfragen zu können. Das sind z.B.:

- aktuell zur Verfügung stehende Partitionen
- aktuell freie bzw. genutzte Partitionen
- Nutzerzugänge und Attribute der Partitionen
- ...

Diese Informationen liefert der Network Resource Manager (NRM). Dieser wird über **nrm** angesprochen. Die Option **-h** liefert eine Übersicht aller möglichen Optionen.
Hier nun die wichtigsten:

nrm -h - Liste aller möglichen Optionen

nrm -pc - Konfiguration der Partitionen anzeigen

nrm -pa - ausführliche Anzeige der aktuell belegten Paritionen

nrm -pm - Anzeige der belegten Partitionen in einer Matrixübersicht

nrm -a <**PARTITION**> - Allokieren einer Partition

nrm -f <**PARTITION**> - Freigeben einer Partition

nrm -r <**PARTITION**> - Rücksetzen einer Partition

nrm -... -q - Operation abbrechen, wenn Partition von einem anderen Nutzer allokiert ist (sonst wird Operation alle 2 Sekunden wiederholt)

nrm -... -1 <**AUSGABEFILE**> - gibt alle stdout - Ausgaben in *AUSGABEFILE* aus

nrm -... -2 <**AUSGABEFILE**> - gibt alle stderr - Ausgaben in *AUSGABEFILE* aus

Gültige Angaben für *PARTITION* sind Partitionsbezeichnungen (erhält man durch Aufruf von **nrm -pc** - sie stehen in der 1. Spalte der Ergebnistabelle) oder Attribute (erhält man durch Aufruf von **nrm -pc** - sie stehen in der 2. Spalte der Ergebnistabelle). Attribute sind dabei festen Partitionsbezeichnungen vorzuziehen, da es immer mehrere Partitionen mit den gleichen Attributen gibt, unter denen der NRM dann wählen kann.
Die Attribute haben dabei folgende Bedeutung:

p16 - 16 Prozessoren

pp16 - 32 Prozessoren (entspricht p32)

kain16 - 32 Prozessoren mit Nutzerzugang über den kain

abel_pfs16 - 32 Prozessoren mit Nutzerzugang über den abel

Die anderen Attribute beziehen sich dann auf entsprechend andere Prozessorzahlen.
Es können von 8 bis 128 Prozessoren in den Stufen 8, 16, 32, 64 und 128 angefordert werden.

Programmaufrufe

Das Starten von Programmen erfolgt im einfachsten Fall durch Aufruf von:

px run -a <**PARTITION**> <**PROG**>

wobei *PARTITION* den oben beschriebenen Angaben entspricht und *PROG* der Name eines für den Parallelrechner übersetzten Programmes ist. Wird bei dem Programmnamen keine Endung angegeben, wird automatisch .px ergänzt. Es ist also nicht erforderlich vorher eine Partition anzufordern, um ein Programm abzuarbeiten.
Ein Beispielprogrammaufruf ist:

px run -a pp32 /home/user/ppcadmin/gcp/parix.test/hello.px

Um die Nutzung des Parallelrechners zu vereinfachen, wird eine grafische Nutzungsoberfläche angeboten. Der Aufruf erfolgt über **xpara** (steht unter /usr/local/bin) auf einem der Frontendrechner. Von dieser Oberfläche aus kann auch auf einfache Weise das Batchsystem DQS bedient werden.

2.2 Aufbau und Funktionsweise von DQS

DQS Version 3.x wurde 1994 am Supercomputer Computations Reasearch Institute[4] der Florida State University entwickelt. Es ist ein Werkzeug zur Nutzung verteilter Ressourcen in einem heterogenen Netzwerk. Durch sogenannte „Komplexe“ kann DQS effektiv an die bestehenden Eigenschaften eines Netzwerkes angepasst werden und kann so die zur Verfügung stehenden Ressourcen effizient benutzen.

2.2.1 Eigenschaften von DQS

DQS versucht, die Anforderungen eines Jobs bestmöglich mit den zur Verfügung stehenden Ressourcen zu erfüllen.

2.2.2 Queue-Komplexe

Queue-Komplexe sind Mengen, bestehend aus Ressourcendefinitionen[7], die mit den Warteschlangen verbunden werden können. Diese Definitionen müssen folgender Form entsprechen:
<Ressourcenbezeichner> [= <Wert>]
<Wert> ::= <Integer> | <String>

Beispiel:
GC
PARTITION = kain32
JOBTIME = 600

Einer Warteschlange können unter dem Punkt *complex_list* verschiedene Komplexe zugeordnet werden. (siehe Anhang B)

2.2.3 Benutzung von DQS

Die DQS Kommandos

Die Kommandos können hier nur kurz in ihrer Funktion erklärt werden. Die komplette und ausführliche Beschreibung steht unter [5].

qsub sendet ein DQS-Skript an den *qmaster*

qdel löscht das entsprechende DQS-Skript

qalter modifiziert ein schon beim *qmaster* stehendes Skript

[7] anstelle von Ressource wird vor allem im DQS-Skript auch der Begriff *Attribut* verwendet

qstat zeigt aktuellen Queue- und Jobstatus von DQS an

qmod erlaubt privilegierten Nutzern (meist Sysop) die Eigenschaften von Warteschlangen zu ändern

qconf Modifikation der DQS-Konfiguration, Queues, Hosts, Nutzerzugangstabellen, Komplexe, ...

Die DQS-Skript-Kommandos

DQS-Skript-Kommandos dienen der Definition zur Jobausführung benötigter Ressourcen, der Steuerung von Ein- und Ausgabeströmen, uvm. DQS-Skript-Kommandos beginnen innerhalb eines Skriptes immer mit **#$**. Darüberhinaus können sie aber auch als Argumente beim Aufruf von **qsub <Job>** übergeben werden. Nachfolgend stehen die wichtigsten Kommandos, die zum jetzigen Zeitpunkt implementiert wurden.

-l <Ressourcenanforderung> definiert eine Ressourcenanforderung der Form:
<Ressourcenanforderung> ::= <Ressourcenbezeichner> [<Operator> <Wert>] und
<Operator> ::= **.lt.** | **.le.** | **.eq.** | **.ge.** | **.gt.** | **.ne.**

.lt. kleiner $<$

.le. kleiner oder gleich $\leq$

.eq. gleich $=$

.ge. größer oder gleich $\geq$

.gt. größer $>$

.ne. ungleich $\neq$

-hard -l <Ressourcenanforderung> definiert, daß diese Ressourcenanforderung vollständig erfüllt werden **muß** bevor der Job gestartet werden kann (z.B.: -hard -l PARTITION.eq.kain32)

-soft -l <Ressourcenanforderung> definiert, daß durch DQS versucht wird diese Ressourcenanforderung bestmöglich zu erfüllen, allerdings gibt es keine Garantie dafür (z.B.: -soft -l JOBTIME.le.100)

-cell <Cell_Name> legt die AFS-„Zelle“ fest

-cwd zur Jobausführungszeit soll das aktuelle Verzeichnis verwendet werden

-e <stderr_Pfade> Standard Error Ausgabe der Form: [<hostname:>] <Pfad> [, [<hostname:>] <Pfad>]

-o <stdout_Pfade> Standard Ausgabe analog **-e**

-j y | n gibt an, ob Fehlermeldungen (*stderr*) mit nach *stdout* geschrieben werden sollen. (Standard: nein)

-N <Name> legt den Jobnamen fest

-passwd_file <Filename> File, in dem sich das (die) AFS-Passwort befindet. Dies wird für die AFS Reauthorisierung verwendet.

2.2.4 Gesamtfunktion von DQS anhand von *qsub*

Der funktionelle Aufbau von DQS wird als DeMarco-Diagramm dargestellt. Der Datenaustausch zwischen den einzelnen Modulen erfolgt über Sockets.

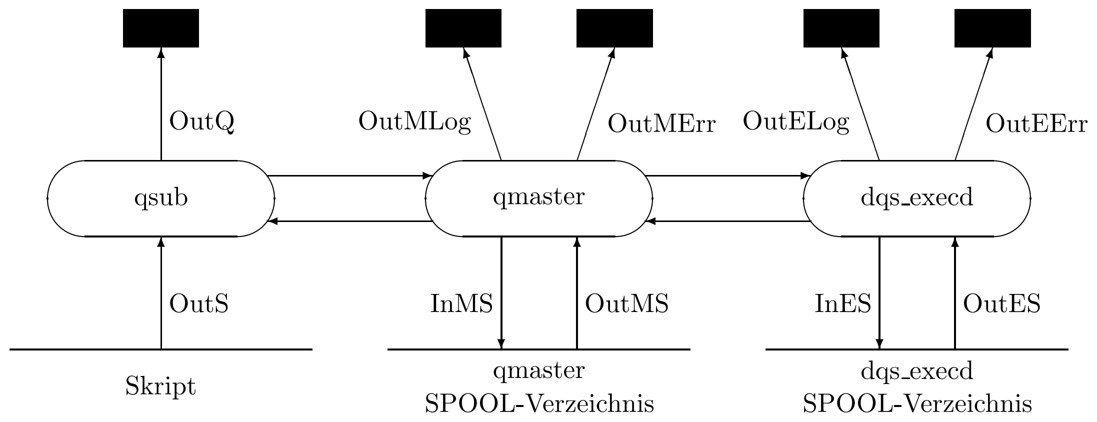

Abb. 1: DeMarco Diagramm der Gesamtfunktion

Erläuterung der Datenströme aus Abb. 1	
OutS	Skriptinhalt , DQS-Job-Kommandos
OutQ	Quittierung
InMS	Queue-, Komplex-, Hostlist-, ACL-, Jobstruktur
OutMS	Queue-, Komplex-, Hostlist-, ACL-, Jobstruktur
OutMLog	Protokollierung
OutMErr	Fehlernachrichten
InES	Jobstruktur
OutES	Jobstruktur
OutELog	Protokollierung
OutEErr	Fehlernachrichten

Im *qmaster SPOOL-Verzeichnis* (.../SPOOL/qmaster) befinden sich die Verzeichnisse zur Speicherung der Jobdaten (job_dir), der Warteschlangenkonfigurationen (queue_dir) sowie im Unterverzeichnis common_dir die Dateien für die Komplexdefinitionen (complex_file), die *trusted_host_list* (host_file) u.a. Es existiert für jeden Prozeß *dqs_execd* ein eigenes *SPOOL-Verzeichnis* unter .../SPOOL/dqs_execd/<Rechner>[8]. Dises nimmt wiederum Verzeichnisse zur Speicherung von Jobzuständen (exec_dir, job_dir) und Auslastungsinformationen (rusage_dir) auf.

[8]Hostname der Maschine, auf der *dqs_execd* gestartet wurde

Zusammenarbeit von qsub, qmaster und dqs_execd

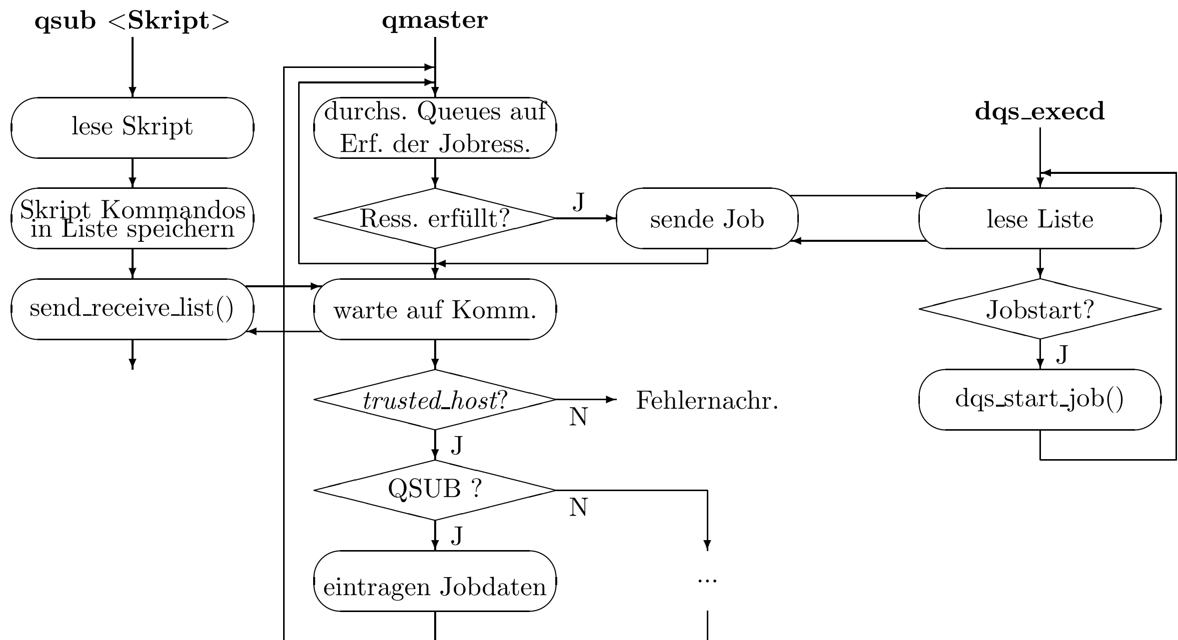

Abb. 2: schematische Darstellung der Funktion von **qsub**, **qmaster** und **dqs_execd**

3 Installation des DQS-Systems, Realisierung notwendiger Anpassungen und praktische Erprobung einer Minimalvariante

3.1 Installation

Die komplette Installationsanleitung wird mit DQS geliefert (.../DQS/DQS_readme.ps) oder man findet sie unter [6]. Nachfolgend wird auf eine Beispielinstallation eingegangen.

3.1.1 Systemanforderungen

DQS wurde auf folgenden Plattformen installiert und getestet:

Hardware	Hersteller	Betriebssystem
alpha	DEC	osf1.3
i486	-	linux
hppa1.1	HP	hpux
mips	DEC	osf1-1.0
mips	DEC	ultrix4.2
mips	SGI	irix4.0.5
mips	SGI	irix5.1.1.1
rs6000	IBM	aix3.2
sparc	SUN	solaris2.3
sparc	SUN	sunos4.1.3
vax	DEC	ultrix4.2

3.1.2 Übersetzung und Konfiguration des Systems

Es existieren zwei Verfahren, um nach erfolgtem Auspacken des DQS-Paketes die Konfiguration des Systems vorzunehmen. Der einfachste Weg ist aus dem Verzeichnis *DQS* heraus **make config** zu starten. Die Installationsdaten werden durch das Konfigurationsprogramm abgefragt und in *DQS/SRC/dqs.h* und *DQS/Makefile* eingetragen. Die andere Möglichkeit ist, diese Daten in den oben genannten Dateien direkt einzutragen. Es sei hierzu nochmals auf [6] verwiesen.

Die einzelnen Konfigurationsdaten sind sehr ausführlich in [6] beschrieben, so daß hier nur auf einen wichtigen Punkt eingegangen werden soll. Da DQS in einem heterogenen Netzwerk arbeitet, wurden verschiedene Sicherheitsmechanismen eingebaut. Der wichtigste hierbei ist die sogenannte *trusted_host_list* - eine Liste von Hostnamen. Nur von den dort eingetragenen Rechnern sind DQS-Kommandos wie z.B. *qsub* benutzbar, da sie nur von diesen Maschinen auf den *qmaster* zugreifen können. Die Hostnamen werden nach dem ersten Start von *qmaster* von einem DQS-Manager eingetragen. Problem hierbei ist, daß der Host auf dem dieses erste DQS-Kommando (**qconf -ah**) ausgeführt werden soll, zu diesem Zeitpunkt bereits in die *trusted_host_list* eingetragen sein muß. Dies geschieht unter dem Punkt **DEFAULT_HOST** im File *DQS/SRC/dqs.h*. Bei

falschem Hostnamen erscheint eine Fehlernachricht, daß der *qmaster*-Dämon nicht kontaktiert werden kann, wobei der richtige Hostname mit im Fehlertext auftaucht! Dieser ist dann in *DQS/SRC/dqs.h* einzutragen und DQS neu zu übersetzen.

DQS/SRC/dqs.h

```
...
/* set to TRUE to disable CONF_FILE and dynamic reconfigs */
#define STATIC_CONFIGURATION              FALSE
#define CONF_FILE                         "/home/urz/fs18/dqs-3.1.1/conf/conf_file"
#define RESOLVE_FILE                      "/home/urz/fs18/dqs-3.1.1/conf/resolve_file
#define KEY_FILE                          "/home/urz/fs18/dqs-3.1.1/conf/key_file"
#define AFS                               TRUE

/* simply set to "err_file" and "log_file" to log separately */
/* eg: qmaster@host dqs_execd@host */
/* OR an absolute path for all into one */

#define ERR_FILE             "/home/urz/fs18/dqs-3.1.1/log/err_file"
#define LOG_FILE             "/home/urz/fs18/dqs-3.1.1/log/log_file"

...

#define MAIL_HAS_SUBJ_LINE                TRUE

/* default DQS Cell managers */
/* DEFAULT_MANAGER2 is optional - comment it out if not needed */

#define DEFAULT_MANAGER                   "tla"
#define DEFAULT_MANAGER2                  "jtr"

/* auto mount directory  - this is optional! */
/* if "-cwd" is set, at exec-time if chdir() fails DQS does the followin:
   a) if (!strncmp(job->cwd,AUTO_MOUNT,strlen(AUTO_MOUNT))) strip AUTO_MOUNT
   b) if (strncmp(job->cwd,AUTO_MOUNT,strlen(AUTO_MOUNT))) prepend AUTO_MOUNT
*/

/*#define AUTO_MOUNT                        "/automount"*/

/* Default DQS Trusted Host */
/* DEFAULT_HOST is optional - comment it out if not needed */
/* but could come in handy of gw'ed machines */

#define DEFAULT_HOST                      "abel-f.hrz.tu-chemnitz.de"
```

Die Übersetzung und Installation erfolgt aus dem Verzeichnis *DQS* heraus mit **make installall**.

Nach erfolgter Installation sind noch die Dateien *conf/resolve_file* und *conf/conf_file* anzupassen. Nun ist mit **qmaster3** der Qmaster-Dämon zu starten. Mit **dqs_execd3** wird der Exec-Dämon auf allen Maschinen, auf denen Warteschlangen installiert werden sollen, gestartet. Diese Rechner müssen natürlich vorher mit **qconf -ah <Hostname>** in die *trusted_host_list* aufgenommen worden sein.

conf/resolve_file

```
# NOTE! blank lines NOT permitted
# NOTE! fields must be separated by one(1) AND ONLY one space #
# 1st field = cell_name
# 2nd field = primary qmaster
# 3rd field = primary qmaster alias
# 4th field = secondary qmaster
# 5th field = secondary qmaster alias
tu-chemnitz.de abel-f.hrz.tu-chemnitz.de abel NONE NONE
```

conf/conf_file

```
QMASTER_SPOOL_DIR        /home/urz/fs18/dqs-3.1.1/SPOOL
EXECD_SPOOL_DIR          /home/urz/fs18/dqs-3.1.1/SPOOL
DEFAULT_CELL             tu-chemnitz.de
RESERVED_PORT            TRUE
QMASTER_SERVICE          dqs1_ur
DQS_EXECD_SERVICE        dqs2_ur
KLOG                     /usr/afsws/bin/klog
REAUTH_TIME              60
MAILER                   /usr/bin/mail
DQS_BIN                  /home/urz/fs18/dqs-3.1.1/bin
ADMINISTRATOR            jtr
DEFAULT_ACCOUNT          SCRI_GENERAL
LOGMAIL                  TRUE
DEFAULT_RERUN            TRUE
DEFAULT_SORT_SEQ_NO      FALSE
SYNC_IO                  TRUE
USER_ACCESS              ACCESS_FREE
LOGFACILITY              LOG_VIA_COMBO
LOGLEVEL                 LOG_DEBUG
MIN_GID                  10
MIN_UID                  10
```

```
MAXUJOBS                10
LOAD_LOG_TIME           60
STAT_LOG_TIME           600
SCHEDULE_TIME           90
MAX_UNHEARD             120
ALARMS                  5
ALARMM                  6
ALARML                  7
```

Anschließend können Queues, Komplexe und Zugriffskontrolllisten angelegt werden[9]. Damit ist die Konfiguration des Systems beendet und der eigentliche Job-Betrieb kann aufgenommen werden.

3.1.3 Die Warteschlangen und Ressourcen des GC

Aus der Spezifikation der Aufgabenstellung (Punkt 6) ist ersichtlich, daß Jobs die eine bestimmte Partitionsgröße überschreiten, nur zu bestimmten Zeiten gestartet werden dürfen. Es bietet sich daher an, für solche Jobs eine eigene Warteschlange anzulegen, die zeitgesteuert entweder mit **qmod -d | -e <Queue>** (de)aktiviert wird, oder indem die Jobressourcen nur zu bestimmten Zeiten in die zugehörigen Komplexe eingetragen werden.
Es ist darüberhinaus sinnvoll, Jobs, die ihre Partitionsanforderungen einerseits über Partitionsbezeichnungen und andererseits über Attribute definieren, in verschiedene Warteschlangen einzureihen. Der Sinn ist, den Komplex der Warteschlange für Jobs, die konkrete physische Partitionen benötigen, zeitlich als Ersten zu aktualisieren, damit evtl. freie Partitionen durch diese Jobs allokiert werden können, und nicht durch Jobs, die nur eine best. Partitionsgröße fordern.
Für den GC wurden folgende Warteschlangen definiert:

Bezeichnung	Anz. gleichzeitig auszufür. Jobs	Queue-nr.	zugeh. Komplex	Bemerkung
gc_physic	8	100	gc_p	Jobs mit *PARTITION.eq.<phys. Bezeichnung>*
gc_attribute	8	101	gc_a	Jobs mit *PARTITION.eq.<Attribut>*
gc_128	1	102	gc_all	Jobs mit Partitionsgröße höher als zur normalen Geschäftszeit belegbar

Die Anzahl gleichzeitig auszuführender Jobs resultiert aus der Forderung, daß DQS zur normalen Geschäftszeit maximal 64 Prozessoren durch Jobs allokieren lassen darf. Da die kleineste Einheit 8 Prozessoren umfaßt, könnten demzufolge maximal 8 Jobs (mit je 8 Prozessoren) gleichzeitig gestartet werden.
Soll ein Job auf dem GC ausgeführ werden so muß mindestens die Ressource *PARTITION* definiert werden (siehe 2.2.3). Die Ressource *JOBTIME* kann ebenfalls definiert werden (Zeitangabe in Minuten!). Sollte dies nicht geschehen

[9] durch DQS-Manager und -Operator mittels **qconf** vorzunehmen

so wird durch den Dämon **partd** standardmäßig der Wert 60 (eine Stunde) eingetragen.

4 Vergleich und Diskussion mit dem „Paderborner System“ Diskussion der Anforderungen aus URZ-Sicht

4.1 Das Projekt Computer Center Software der Uni Paderborn

Das Projekt „Computer Center Software [7]“ (CCS) ist eine Software, die die Benutzung von Parallelrechnern wie auch von herkömmlichen Rechnern und zwar transparent für den Anwender gestattet.Darüberhinaus soll der Nutzer so wenig wie möglich mit hardwareseitigen Details der einzelnen Maschinen belastet werden, d.h. CCS trennt die Anwender- von der Betriebssystemschicht (z.B. PARIX). Der Benutzer der unter der sog. Mastershell arbeitet spezifiziert seine Ressourcenanforderungen mittels RDL (Ressource Description Language [8][9]).

4.2 Vergleich der Systeme

Der Vergleich basiert auf der zur Verfügung stehenden Beschreibung von CCS[7].

Eigenschaften	CCS	DQS
gleichzeitige interaktive GC-Nutzung unter PARIX	nein	ja, da DQS auf PARIX aufgesetzt wurde
GC-Steuerung durch	CCS	PARIX
Ressourcendefinition über	RDL	DQS-Job-Kommandos
Kommunikation	RPC	Sockets
AFS-Unterstützung	nein	ja
PVM-Unterstützung	hierzu lag keine Inf. vor	ja
Unterstütung anderer Hardware	bis jetzt nur GC	siehe Punkt 3.1.1

5 Ausbau der auf DQS basierenden Technologie

5.1 Einleitung - ein DQS-Beispieljob

```
# Definition der Partitionsgroesse - 16 Prozessoren
#$ -hard -l  PARTITION.eq.p16

# geschaetzte Jobausfuehrungszeit - 1 Minute
#$ -soft -l  JOBTIME.eq.1

# stdout -> ausgabe
#$ -o ausgabe

# Aufruf des PARIX-Programmes
# dqs_run ersetzt run von PARIX (siehe 2.1)
dqs_run -a p16 hello.px
```

Es ist zwingend, in einem DQS-GC-Job die Partitionsgröße (PARTITION) zu definieren, da der Dämon **partd** dies zur Unterscheidung von GC und anderen DQS-Jobs benötigt. Wird die Jobausführungszeit nicht angegeben, so wird der in **defs.h** definierte Standardwert *DEFAULT_JOBTIME* benutzt.
Darüberhinaus muß **run** durch **dqs_run** ersetzt werden, damit bei fehlgeschlagener Allokierung der Job wieder in die entsprechende Warteschlange eingereiht werden kann[10].

Funktionsweise

Das Skript wird mit **qsub <Skript>** an den **qmaster** gesendet. Dort befindet es sich unter **SPOOL/qmaster/<Host>/job_dir/<JOB_ID>** solange nicht die Ressourcenanforderung *PARTITION.eq.p16* erfüllt werden kann. Der zu entwickende Dämon ermittelt parallel dazu zyklisch den Belegungszustand des GC und trägt die Ressourcendefinitionen freier Partitionen in die zugehörigen (s.u.) Komplexe ein. Sollte dabei die Ressourcenaforderung *PARTITION.eq.p16* des Jobs erfüllt werden, sendet **qmaster** diesen an den entsprechenden **dqs_execd** (siehe hierzu auch Anhang B - Queuedefinitionen - Punkte *hostname* und *complex_list*). Dort wird die Standardausgabe nach *ausgabe* umgeleitet und der eigentliche Job (idF. Kommando **dqs_run ...**) gestartet. Sollte eine gewünschte Partition *p16* noch frei sein und wenn der damit insgesamt von DQS belegte Platz einen bestimmten Wert (siehe 1.1) nicht überschreitet, wird durch **dqs_run** das PARIX-Programm **hello.px** gestartet. Sollte dies nicht der Fall sein, so muß der Job wieder in die entsprechende Warteschlange bei **qmaster** eingereiht werden.

[10] m.H. des Programmes **qrerun**

5.2 partd - Dämon zur Partition- und Jobverwaltung

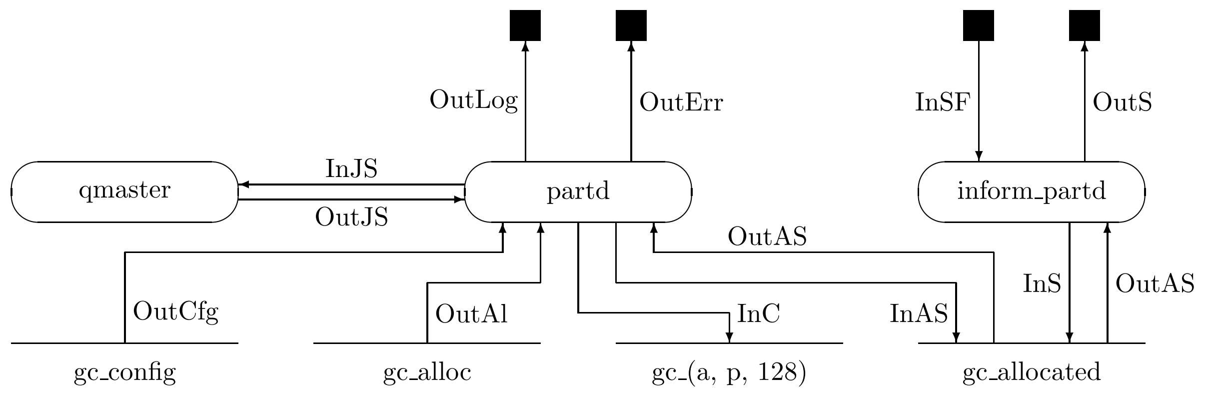

Abb. 3 - DeMarco Diagramm des Dämonen *partd*

Erläuterung der Datenströme aus Abb. 2	
InSF	Größenangabe und Funktion (alloc, free)
InJS	Jobstruktur (Simulation von **qstat**)
InC	Ressourcendefinitionen
InAS	belegte und maximale GC-Größe, Wartezeit (*allocsize*-Struktur)
InS	belegte GC-Größe
OutLog	Informationen über GC-Auslastung, Job-Zeitüberschreitung
OutErr	Fehlerprotokollierung
OutS	belegte GC-Größe
OutJS	Jobstruktur (Simulation von **qalter**)
OutCfg	GC-Konfiguratinsbeschreibung (**nrm -pc**)
OutAl	GC-Allokierungsdaten (**nrm -pa**)
OutAS	belegte und maximale GC-Größe, Wartezeit (*allocsize*-Struktur)

5.2.1 Funktionsweise von partd

Grob gesagt läßt sich der Dämon in zwei Teile zerlegen:

1. den Initialisierungteil, der die partd- und GC-Konfiguration (über **conf_file** und **nrm -pc**) ausliest, und das eigentliche Programm im Hintergrund startet

2. den Teil, der zyklisch die entsprechenden Komplexe modifiziert, laufende Jobs überwacht und die Priorität wartender Jobs verändert (*dqs_run()*

Das Programm partd.c

```
#define MAINPROGRAM

#include "daemon.h"
```

```
main()
{
  daemon partd;

  partd.init();                    /* Initialisierungsteil */
  partd.run();
}
```

Die Funktion *daemon::run()* stellt einen endlosen Zyklus dar, in dem überprüft wird, ob die Warteschlange **gc_128** aktiviert werden kann (außerhalb der Geschäftszeiten) und in dem die entsprechenden Komplexe modifiziert werden. Weiterhin werden die Jobs daraufhin untersucht, ob sie für die Ausführung auf dem GC bestimmt sind, d.h. ob die Ressourcen *PARTITION* und evtl. *JOBTIME* definiert wurden. Ist dies der Fall, so werden diese Jobs, falls sie auf ihre Ausführung warten in ihrer Priorität aktualisiert und anderenfalls ihre Laufzeitvorgabe (JOBTIME) überwacht.
Um die Software möglichst universell einsetzbar und die damit verbundenen Modifikationen so einfach wie möglich zu machen, wurden alle wichtigen Programmparameter in der Datei **defs.h** (siehe Anhang A.1) definiert. Die weiteren Ausführungen beziehen sich teilweise auf diese Konstanten.

5.2.2 verwendetete Dateien und ihr Inhalt

Datei	beinhaltet
	Quelldateien
Makefile	
allocsize.c	Methoden der Klasse *allocsize*
array.c	Methoden der Klasse *array*
tables.c	Klassen *parttable*, *attribtable* und *sizetable*
complex.c	Methoden der Klasse *complex*
config.c	Methoden der Klasse *config*
daemon.c	Methoden der Klasse *daemon*
partd.c	Hauptprogramm für **partd**
inform_partd	Hauptprogramm **inform_partd**
dqs_run	Shellskript für die Ausführung von Jobs auf dem GC unter DQS
conf_file	Beispielkonfiguration des Dämonen **partd**
	nach der Kompilierung
partd	Partition-Dämon
inform_partd	Informationsprogramm für **partd** (s.u.)

5.2.3 die verwendete Klassenhierarchie

Der Dämon wurde vollständig objektorientiert entwickelt. Der Grund dafür war einerseits die Wiederverwendbarkeit von Code wie dies in der Klasse *array* und den daraus abgeleiteten Klassen *...table* benötigt wurde. (siehe Abb. 4 und Anhang A.2 ff) Andererseits wurde durch die, mit der Objektorientierung einhergehende, Datenkapselung eine einfache Wartung der Software erreicht.

Eventuell notwendigen Änderungen können lokal, meist innerhalb einer einzigen Klasse, durchgeführt werden.

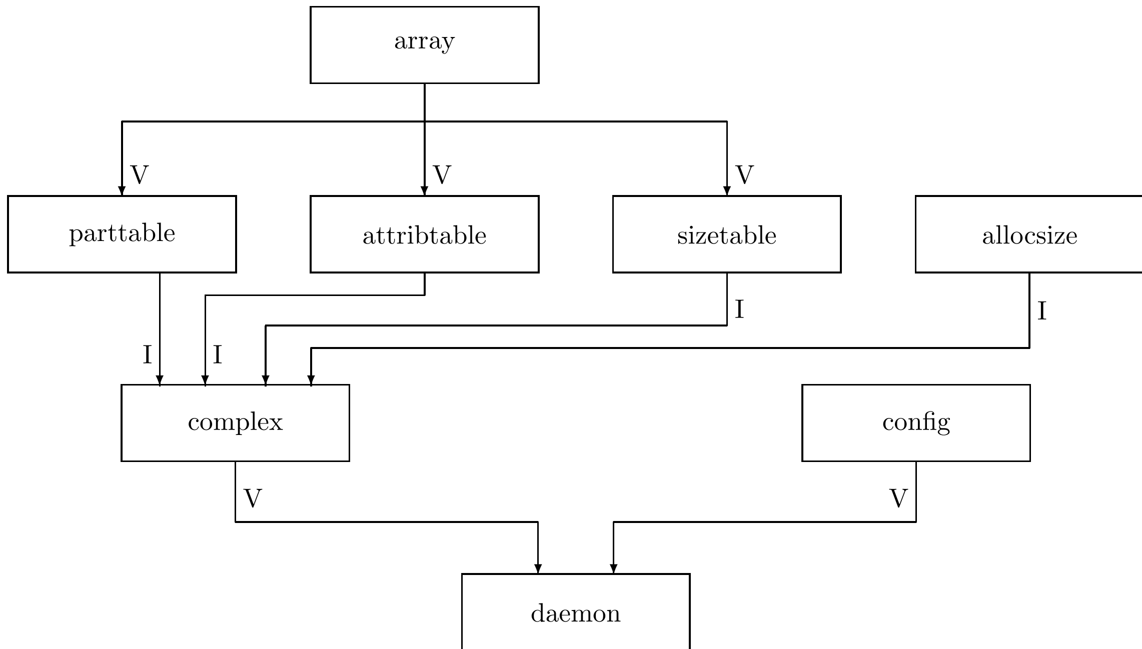

Abb. 4 die zugrunde liegende Klassenhierarchie

Beziehung	Bedeutung
A $\xrightarrow{V}$ B	B geht aus A durch Vererbung hervor
A $\xrightarrow{I}$ B	eine Instanz von A ist in B enthalten

5.2.4 funktionelle Anforderungen an die Klassen

allocsize stellt grundlegende Funktionen zur Verwaltung eines gemeinsam genutzten Objektes (in Datei ALLOCSIZE_FILE) zur Speicherung der u.a. aktuell von DQS belegten Partitionengröße für die Klasse *complex* sowie das Unterstützungsprogramm **inform_partd** bereit (Anhang A.4)

array liefert grundlegende Funktionen zur Verwaltung von Feldern wie z.B.: suchen oder einfügen und ist Basisklasse für die Klassen *...table* (Anhang A.2)

parttable Funktionen zur Verwaltung eines Feldes von Partitionseinträgen, d.h. Speicherung der Namen, Größen, und Attribute der einzelnen Partitionen (Anhang A.3)

attribtable Funktionen zur Verwaltung eines Feldes von Attributen (Bereitstellung des Komplexes *gc_a*) (Anhang A.3)

sizetable Funktionen zur Verwaltung eines Feldes von Größenangeben (Bereitstellung des Komplexes *gc_a*) (Anhang A.3)

complex modifiziert ausgehend vom aktuellen GC Belegungszustand die entsprechenden DQS-Komplexe (Anhang A.5)

config liest die Konfigurationsdaten wie z.B.: tägliche Geschäftszeiten, maximale GC-Größe, maximal von DQS belegbare Größe, u.a. ein (Anhang A.6)

daemon modifiziert zeigesteuert Prioritäten wartender Jobs, überwacht laufende Jobs auf Zeitüberschreitung und modifiziert die einzelnen Komplexe (Anhang A.7)

5.2.5 Scheduling-Strategien

Da jedem Job die erforderliche Partitionsgröße und die geschätzte Jobausführungszeit „mitgegeben" werden, ist es möglich ein deterministisches Scheduling durchzuführen. Aus der Menge wartender Jobs können also diejenigen ausgewählt werden, die den GC optimal ausnutzen. Der Scheduler würde in diesem Falle ein Optimierer sein.
Allerdings wird der GC nicht nur von DQS, sondern auch von Anwendern benutzt, die ihre Jobs direkt über PARIX starten. Darüberhinaus kann auch nicht abgeschätzt werden ob die Jobausführungszeit eingehalten wird. Diese Aspekte belegen, daß das deterministische Verfahren zum jetzigen Zeitpunkt zu ineffizient ist, da eine Optimierung der GC-Ausnutzung nicht gewährleistet werden kann. Demzufolge muß ein anderes Schedulingverfahren gefunden werden.
Da DQS aus der Menge der wartenden denjenigen zuerst startet, der die höchste Priorität hat, muß der Scheduler diese Priorität für die entsprechenden Jobs in deren Jobstruktur eintragen und sie an den qmaster senden. Dieses Verfahren auf dem dem im Punkt 3.2.2 beschriebenen auf.
Aus der Aufgabenstellung ist ersichtlich, daß in die Berechnung der Priorität folgende Aspekte einfließen sollen:

Partitionsgröße	je größer desto geringere Priorität
Jobausführungszeit	je kürzer desto höhere Priorität

Würden diese zwei Daten allein für die Prioritätenberechnung genutzt werden, so wäre die Gefahr des „Verhungerns" gegeben. Ein Job mit großer Jobausführungszeit und Partitionsgröße käme gegenüber vielen Jobs mit kleiner Jobausführungszeit und Partitionsgröße nie zum Zuge. Um dies zu verhindern muß in die Berechnung auch die Wartezeit des Jobs mit einfließen und zwar in der Form:

je länger die Wartezeit, desto höher die Priorität

Daraus kann man eine erste einfache Berechnungsvorschrift für die Priorität eines wartenden Jobs ableiten.

$$Priorität_1 = \frac{Wartezeit}{Partitionsgröße * Jobausführungszeit}$$

Natürlich kann ebenso eine andere Funktion gewählt werden, durch das modulare, objektorientierte Konzept des Dämonen braucht hierzu nur die Funktion *new_priority()* aus der Klasse *daemon* modifiziert zu werden.
Um der Anforderung, daß Jobs mit sehr hoher Partitionsgößenforderung (Job_2), nur zu bestimmten Zeiten und da bevorzugt bearbeitet werden sollen, gerecht zu werden, wird für diese Jobs eine modifizierte der obigen Formel verwendet.

$$Priorität_2 = \frac{Wartezeit}{Partitionsgröße * Jobausführungszeit} + const$$

Dabei muß durch eine Beschränkung von $Priorität_1$ für $\forall Job_1$ auf maximal $const$ dafür gesorgt werden, daß

$$\forall Job_1, Job_2 \in Jobs : Priorität_1(Job_1) < Priorität_2(Job_2)$$

gilt.

5.2.6 Konfigurationsdaten

Die Konfigurationsdaten erlauben die Modifikation von wichtigen Konstanten die sich evtl. im laufe der zeit ändern können, ohne daß dabei eine Neuübersetzung, sondern nur ein Neustart von **partd** nötig ist. Diese Änderungen könenn an der Konfigurationsdatei *CONFIG_FILE* (**conf_file**) vorgenommen werden. Folgende Daten können in der Konfigurationsdatei **conf_file** enthalten sein:

business_time definiert die Geschäftszeiten an bestimmten Tagen

> **business_time** <Tag> [**-** <Tag>] **:** <Zeit> [**-** <Zeit>] [**,** <Tag> ...
>]
> Tag ::= **MON** | **TUE** | **WED** | **THU** | **FRI** | **SAT** | **SUN**
> Zeit ::= (0...24)
>
> (Bsp.: **business_time mon-fri: 7-18, sat-sun: 0**) Zeit = 0 bedeutet daß für die angegebenen Tage keine Geschäftszeiten existieren.
>
> Die Standardeinstellung beträgt für Montag bis Freitag *BUSINESS_TIME_START* und *BUSINESS_TIME_STOP* (defs.h).

ressource_partition definiert die Ressourcenbezeichnung, die für die Angabe der gewünschten Partitionsgröße verwendet wird (Standard:

RESS_PARTITION)

ressource_partition <Bezeichnung>

ressource_jobtime Ressourcenbezeichnung für die Jobausführungszeit (Standard: *RESS_JOBTIME*)

ressource_jobtime <Bezeichnung>

gc_size gibt die maximal verwendbare GC-Größe an (Standard: *GC_SIZE*)

gc_size <Größe>

max_alloc_b_time gibt die maximal verwendbare GC-Größe während der Geschäftszeiten an (Standard: *MAX_ALLOC_B_TIME*)

max_alloc_b_time <Größe>

log_file definiert die Datei in die Mitteilungen, z.B. über Laufzeitüberschreitungen geschrieben werden sollen

err_file definiert die Datei in die alle Fehlermeldungen des Dämonen geschrieben werden sollen

Ein Beispiel für eine Konfigurationsdatei:

```
# normale Geschaeftszeiten (Wochenende nichts)
business_time mon-fri: 7-18, sat-sun: 0

# waehrend dieser Zeit nur insgesamt max. 32 Prozessoren belegen
max_alloc_b_time 32

log_file logfile
err_file errfile
```

5.3 Das Informationsprogramm für die Jobs - inform_partd

Das Programm **inform_partd**, welches von **dqs_run** (siehe 5.4 Listing) heraus aufgerufen wird, hat den Zweck eine bestimmte Partitionsgröse über den partd entweder zu allokieren oder freizugeben.
Syntax:

inform_partd a|f <Größe>

Falls die Allokierung fehlschlägt, weil z.B.: die DQS zur verfügung stehende GC-Größe erschöpft ist, wartet das Programm eine in der Datei **gc_allocated** definierte Zeit um danach die Kontrolle wieder an **dqs_run** abzugeben, welches dann den Job „restartet“.

5.4 notwendige Erweiterung von DQS

Im Verlauf des Projektes wurde die Notwendigkeit der Erweiterung von DQS[11] erkannt, da wichtige Funktionen und Programme zu diesem Zeitpunkt nicht implementiert waren, obwohl die Dokumentation von DQS darüber gegenteiliges aussagte. Diese Funktionen mußten also nachträglich implementiert werden und zwar in der Weise, daß eine Kompatibilität zu nachfolgenden DQS-Versionen, die diese Möglichkeiten besitzen, möglichst erhalten blieb.
Folgende Funktionen mußten implementiert werden, damit das Projekt fortgesetzt werden konnte:

qconf -mc <Komplex> [Datei] wird für die Modifikation der einzelnen Komplexe verwendet

qrerun <Jobidentifikator> benötigt für den Neustart von Jobs nach erfolgloser Partitionsallokierung

Die Funktion **qconf -mc <Komplex>** wird von den Methoden der Klasse *complex* zur Modifikation der drei verschiedenen Komplexe benötigt. Das Programm **qrerun** wird in dem Programm **dqs_run** für einen evtl. nötigen Jobrestart verwendet.

Programm dqs_run (Ausschnitt)

```
...
#---------------RunMPP--------------------------------------------------
# Name: RunMPP
# Purpose: execute a parix binary on an MPP machine's partition
# Arguments: $Args : Parix application arguments

RunMPP()
{
    if [ $System = a ] ; then
# ignore all signals before calling nrm, otherwise partitions
# could remain allocated
trap "" $ActSignals
if [ $quiet = 0 ] ; then
    echo >&2 "$MyName : Requesting network by calling nrm."
fi
        # Aufruf nrm
# nrm returns: PhysPartName x y z LinkMap HostName HostLink HostProcLink Proc Link ...
NRMresult=`$PARIX_OSBIN/nrm $Attributes $EntryNames $nrmOpts`
    else
if [ $quiet = 0 ] ; then
    echo >&2 "$MyName : Reading partition info from file $ConfFile."
fi
```

[11] der derzeit aktuellen Version 3.1.1

```
NRMresult=`cat $ConfFile`
    fi
    result=$?
    if [ $verbose = 1 ] ; then
echo >&2 "$MyName : Partition info is $NRMresult."
    fi
    if [ $result = 0 ] ; then
set ${NRMresult:-""}
AtExit CleanMPP
        # Partitionsbezeichnung
PartitionName=$1
        # Groessenangaben
NRMDimX=$2
NRMDimY=$3
NRMDimZ=$4

if [ "$DimX" = "UseDefault" ]; then
    DimX=$NRMDimX
    DimY=$NRMDimY
    DimZ=$NRMDimZ
fi

LinkMap=$5
shift 5

Entries=$*

        # allokiere bei partd
        /home/urz/fs18/dqs-3.1.1/partd/inform_partd a $NRMDimX $NRMDimY 2
        if [ $? = -1 ] ; then
          # kein ausreichender Platz
          /home/urz/fs18/dqs-3.1.1/partd/inform_partd f $NRMDimX $NRMDimY 2
          # Job-Neustart
          /home/urz/fs18/dqs-3.1.1/bin/qrerun $JOB_ID
        else
          RunFixedOrMPP
        fi
    else
      # Allokierungsfehler bei nrm -> Job-Neustart
      /home/urz/fs18/dqs-3.1.1/bin/qrerun $JOB_ID
    fi
}
...
```

5.4.1 Analyse der Funktion von DQS

Alle DQS-Programme (wie z.B. qconf, qsub, qstat, ...) sind in ihrem Aufbau zweigeteilt, einmal das eigentliche Programm und der „Ausführungsteil“, der Bestandteil des **qmasters** ist. Die Aufgabe des DQS-Programmes besteht nur darin, die ihm übergebenen Daten auf Korrektheit hin zu überprüfen, diese in eine bestimmte Form (s.u.) zu bringen und sie an den qmaster zu senden. Danach können durch den qmaster eventuell zurückgelieferte Daten verarbeitet werden.
Der qmaster-Prozeß ermittelt anhand der empfangenen Daten, die zugehörige Funktion und startet diese. Das bedeutet, daß bei Kenntnis des Aufbaus der Kommunikationsstrukturen diese Funktionen ebenfalls von anderen Programmen aus genutzt werden können.

5.4.2 Die Kommunikation zwischen DQS

Wie schon gesagt basiert die gesamte DQS-Kommunikation auf sockets. DQS stellt hierzu eine Reihe von Fuktionen zur Verfügung, von denen hier drei vorgestellt werden sollen:

dqs_send_list(char *cell, char *service, int sfd, dqs_list_type *head)
dqs_get_list(char *sfd, dqs_list_type **head)
dqs_send_receive_list(..., dqs_list_type *out, dqs_list_type *in)

Während die Argumente *cell*, *service*, *sfd* für Kommunikationsaufbau und *sim*steuerung verwendet werden, sind die eigentlich zu übertragenden Daten in den Argumenten vom Typ *dqs_list_type ** enthalten. Dieses Feld (definiert in struct.h) erlaubt außer der Speicherung von Identifikations- und Aktionsdaten (in den Variablen *who* und *type*) auch die von Queue- und Jobstrukturen. Darüberhinaus kann über die drei letzten Variablen eine verkettete Liste erzeugt werden, wie dies von einigen DQS-Funktionen (qstat) praktiziert wird. Diese Struktur ist die einzige, die für die Kommunikation der verschiedenen DQS-Programme untereinander benutzt wird. Die Elemente *type* und *int0* klassifizieren hierbei die gewünschte Funktion die im **qmaster** ausgelöst werden soll. Die Inhalte der anderen Elemente sind allein hiervon abhängig.

```
struct  listel {
        u_long32           who;
        u_long32           type;
        u_long32           status;
        u_long32           int0;
        u_long32           int1;
        u_long32           int2;
        u_long32           int3;
        u_long32           bufsize;
        char               *user;
        char               *str0;
        char               *str1;
```

```
    char               *str2;
    char               *str3;
    char               *buf;
    dqs_rusage_type    *rusage;
    dqs_queue_type     *queue;
    dqs_conf_type      *conf;
    dqs_job_type       *job;
    dqs_me_type        *me;
    struct listel      *chain;
    struct listel      *tid;
    struct listel      *next;
  };

typedef struct listel dqs_list_type;
```

Das heißt, daß auch andere Programme sich dieser DQS-Funktionen bedienen können, indem sie Identifikations- und Aktionsbezeichnungen realer DQS-Programme in der *listel*-Struktur eintragen und diese den Kommunikationsfunktionen übergeben. Diese Vorgehensweise wird auch bei der Implementation von **qrerun** und **partd** benutzt. Die hierbei notwendigen Informationen, über die Inhalte der einzelnen Strukturelemente, mußten dabei durch Analyse der Quelltexte der zu simulierenden Funtionen gewonnen werden, was eine sehr zeitaufwendige Arbeit darstellte.

5.4.3 Modifikation von qconf

Zur Implementation der Funktion *modify complex* war es notwendig folgende Änderungen an den DQS-Quelldateien vorzunehmen.
In der Datei **SRC/globals.h** mußte bei der Definition des Feldes *dqs_options* unter *mc_OPT* eine 1 eingetragen werden, damit **qconf** diese Option erkannte.

```
...
/*            n    q    q    q    q    q    q    q    q    q    q    q    q    q
              o    a    c    d    h    i    m    m    m    m    r    r    s    s
              n    l    o    e    o    d    a    o    o    s    e    l    e    h
              e    t    n    l    l    l    s    d    v    g    r    s    l
                   e    f         d    e    t         e         u         e
                   r                        e                   n         c
                                            r                             t


*/
...
/* mc_OPT */  0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
...
```

Als nächstes wurde die Datei **SRC/dqs_parse_qconf.c** modifiziert, so daß die **qconf -mc** übergebenen Argumente korrekt ausgewertet werden konnten.

Daraufhin konnte in **SRC/dqs_add_del.c** die Funktion *dqs_modify_complex()* implementiert werden, die letztendlich die Modifikation eines Komplexes durchführt.

SRC/dqs_add_del.c

```
...
int dqs_modify_complex(c)
dqs_list_type *c;

/*

  dqs_modify_complex - modifies the Complex_list and to Complex_hash

  if the invoking process is the qmaster the complex list is spooled
  to disk.

  returns:
  -1 if complex not exists;
  0  if successful
  1  on error

  */

{

     dqs_list_type *lp;
     dqs_list_type listel;
     dqs_hash_type *hp;

     DENTER_EXT((DQS_EVENT,"dqs_modify_complex"));

     if (!Complex_hash)
     Complex_hash=dqs_hash_init();

     if (!c)
     {
          ERROR((DQS_EVENT,"error: NULL ptr passed to dqs_modify_complex()"));
          DEXITE;
          return(1);
     }

     hp=dqs_hash_lookup(c->str0,Complex_hash);
/* Komplex vorhanden ? */

     if (!hp)
     {
```

```
        ERROR((DQS_EVENT,"error: the complex \"%s\" does not exist",
               c->str0));
        DEXITE;
        return(-1);
    }

    dqs_hash_del(c->str0,Complex_hash);
/* loesche alten Komplex */
    dqs_hash_add(c->str0,Complex_hash); /* neuen eintragen */

    bzero(&listel,sizeof(listel));
    listel.str0=dqs_string_insert(NULL,c->str0);
    listel.chain=c->chain;
    c->chain=NULL;

    Complex_head=dqs_del_str0(Complex_head,c->str0);
    Complex_head=dqs_insert(DQS_STR0,ASCENDING,Complex_head,&listel);

    hp=dqs_hash_lookup(c->str0,Complex_hash);
    if (!hp)
    {
        CRITICAL((DQS_EVENT,"error: Complex_hash is screwed"));
        abort();
    }

    lp=dqs_locate_str0(Complex_head,c->str0);
    hp->vp=(char *)lp;

    if (me.who==QMASTER) /* Komplex schreiben */
    dqs_write_list_to_disk(NULL,COMPLEX_FILE,Complex_head,ALL);

    /*dqs_show_complex_list(Complex_head);*/

    DEXIT;
    return(0);
}
...
```

Diese Funktion wurde schließlich noch in die Datei **SRC/func.h** eingetragen. Die so erfolgte Ergänzung ist völlig kompatibel zu späteren Versionen von DQS, d.h. sollte eine nachfolgende DQS-Implementation diese Fähigkeit besitzen kann sie sofort an Stelle der jetzigen veränderten Version benutzt werden.

5.4.4 Implementation von qrerun

Die Idee bei der Entwicklung von **qrerun** war, erstens das DQS-Programm **qstat** zu wie oben beschrieben zu simulieren, aus den zurückgelieferten Job-

strukturen die Neuzustartende zu finden und diesen Job mittels einer Simulation von **qsub** wieder in die Warteschlangen einreihen zu lassen.

Datei qrerun.c (Ausschnitt)

```
...
     DENTER_MAIN((DQS_EVENT,"qstat"));  /* simuliere QSTAT */

     dqs_setup(QSTAT,argv[0]);
     dqs_setup_sig_handlers();

     bzero(&listel,sizeof(listel));     /* hole u.a. Jobstrukturen */
     listel.type=QSTAT;
     if (dqs_send_receive_list(me.default_cell,conf.qmaster_service,&listel,&resp) < 0
     {
          DEXITE;
          exit(DQS_EAGAIN);
     }
...
     lp=resp->next;                     /* Jobstrukturen liegen unter 'next' */
     while (lp)
     {
          (void) dqs_add_job(lp->job);  /* speichern der Jobstrukturen lokal */
          lp=lp->next;
     }

     lp=Job_head;                       /* alle Jobstrukturen durchsuchen */
     while (lp && ok)
     {
        if (! strcmp(lp->job->dqs_job_name, argv[1]))   /* Jobident. gefunden ? */
        {
                ok = 0;
                bzero(&listel,sizeof(listel));
                listel.job = lp->job;                   /* speichere Jobstruktur */

                dqs_get_passwd_info(listel.job);

                old_who=me.who;                         /* simuliere QSUB */
                me.who=QSUB;
                listel.type=QSUB;
...                                                     /* sende Job ab */
                dqs_send_list(me.default_cell,conf.qmaster_service,sfd,&listel);
...
```

Es bleibt anzumerken, daß trotz des anscheinend einfachen Aufbaus des qrerun-Programmes die Analyse der zugrundeliegenden Kommunikationstechniken und der Speicherverfahren (Hash-Listen) einen erheblichen Zeitaufwand darstellten,

der bei „ehrlicher“ Dokumentation von DQS zwar nicht vermeidbar aber doch zumindest vorher abschätzbar gewesen wäre.

5.5 Installation des Dämonen

Die momentan aktuelle Version des Dämonen ist 1.01. Nach Entpacken des Archives partd-1.01.tgz, welches sich unter /home/urz/fs18/partd-src/ auf dem Rechner abel befindet, können in **defs.h** und **Makefile** Anpassungen an die jeweiligen Gegebenheiten durchgeführt werden. Es ist darauf zu achten, daß der Pfad zu den DQS-Include-Dateien und zu der DQS-Bibliothek **dqs.a** korrekt gesetzt wurde. Mit

```
make all
```

werden die Programme übersetzt. Danach kann gegebenenfalls die Konfigurationsdatei *CONFIG_FILE* angepaßt werden. Zur Ausführung benötigt der Dämon root-Rechte, falls DQS über reserved-Port kommuniziert.

6 Integration der DQS-Technologie des Parallelrechners in die URZ-Technologie

6.1 DQS-Systemadministration

In der momentanen Installation befinden sich sowohl **qmaster** wie auch **dqs_execd** und **partd** auf einer Maschine[12]. Für eine Integration in das schon testweise URZ-weit laufende DQS müssen die Warteschlangen *gc_physic*, *gc_attribute* und *gc_128* beim URZ-qmaster[13] eingetragen werden. Danach muß der in diesen Warteschlangen angegebene Frontendrechner in die *trustet_host_list* mit **qconf -ah ...** aufgenommen und auf diesem die Dämonen **dqs_execd** und **partd** gestartet werden.

Somit ist die Integration in die URZ-Technologie ohne großen Aufwand möglich.

6.2 Sicherheitsaspekte

Es ist abschließend nötig auf einige Sicherheitsaspekte einzugehen. Da der Dämon **dqs_execd** Jobs mit den Rechten der, in der zugehörigen Jobstruktur definierten, zugehörigen Nutzer starten soll, benötigt er hierzu zwingend root-Rechte. Dies wiederum zieht nach sich, daß die socket-Kommunikation über *reserved-ports* vonstatten geht, da ansonsten jeder, der die Zeit aufbringt die DQS-Job-Struktur zu analysieren, in der Lage wäre, Jobs mit fremden Nutzerrechten auszuführen. Die notwendige Konsequenz ist, daß alle DQS-Programme und die Dämonen über *reserved-ports* kommunizieren, d.h. sie alle müssen mit root-Rechten laufen.

Ein weiteres Sicherheitsproblem ist es, den Nutzer *root* als DQS-Manager einzutragen, denn in diesem Fall könnte jeder, der auf einer Maschine, die in die *trustet_host_list* eingetragen ist, ein root-Passwort besitzt, Veränderungen am gesamten DQS-System vornehmen (z.B.: Zugriffskontrolllisten ändern). Diese Manipulationen könnten trotz Abrechnung durch DQS nur schwer verfolgt werden, da zwar bekannt wäre, daß *root* einen Eingriff vornahm, aber keinerlei Informationen existieren von welcher Maschine aus. Das heißt der Nutzer *root* sollte keinesfalls als DQS-Manager definiert werden!

Des weiteren ist die sog. AFS-Reauthorisierung zu nennen. DQS speichert das AFS-Passwort des Nutzers (verschlüsselt!), um bei bei Ablauf der Zeitspanne[14] einen Kind-Prozeß zu erzeugen und mittels des gespeicherten Passwortes eine Reautorisierung vorzunehmen.

Um Zugriffsbeschränkungen zu bestimmten Queues zu realisieren stellt DQS für jede Warteschlange die *user_acl* und *xuser_acl* zur Verfügung, über die definiert werden kann, welche Nutzer Zugriff auf diese Warteschlange haben, bzw. welche nicht. Diese Listen werden in der Warteschlangendefinition eingetragen.

[12] Frontendrechner: abel.hrz.tu-chemnitz.de

[13] auf pyrrhus.hrz.tu-chemnitz.de

[14] ein AFS-Account ist nur über eine best. Zeitspanne gültig

6.3 Ausblicke zu DQS und partd

Als erstes muß hier die Integration der DQS-Technologie für den Parallelrechner in das schon testweise URZ-weit laufende DQS genannt werden. Dies wird die zuerst wichtigste Aufgabe sein.
Parallel dazu findet z.Z. auch die Entwicklung von Oberflächen zur allgemeinen Benutzung von DQS und des Parallelrechners z.B. unter X11 (xpara) statt. Damit wird der Anwender von der Erstellung der DQS-Shell-Skripte entlastet, er braucht nicht die einzelnen DQS-Job-Kommandos zu erlernen, was natürlich den Umstieg auf DQS erleichtert.
Darüberhinaus stellt **partd** natürlich kein abgeschlossenes Softwaresystem dar. Die Funktion *new_priority()* sowie verschiedene Konstanten (z.B. Geschäftszeitvorgaben) können sich im Laufe der Nutzung des GC durch DQS-Jobs als unzweckmäßig erweisen, so daß eine Modifikation der Konfigurationsdatei oder der Software selbst nötig würde. Durch das zugrundeliegende Programmiermodell dürften diese Änderungen aber auch von einem, nicht DQS und **partd**-erfahrenen, Programmierer bzw. Operator ohne Probleme ausgeführt werden können.

A Header-Files des Dämonen partd

A.1 wichtige Voreinstellungen - defs.h

```
/* Def's fuer die verschiedenen Klassen */

#define BUFSIZE               512

/* class array */

#ifdef ARRAY
  #define FOUND         1       /* von find() benutzt */
  #define NOT_FOUND     0
#endif


#ifdef TABLES

  /* class parttable */

  #define PT_MAXENTRIES         128     /* max. Eintraege in part_table */
  #define PT_NAMELENGTH         32      /* max. Laenge des Partitionnamens */
  #define PT_ATTRIBLENGTH       64      /* max. Laenge des Attribut-Feldes */

  /* class attribtable */

  #define AT_MAXENTRIES PT_MAXENTRIES / 2       /* max. Eintraege in attrib_table */
  #define AT_LENGTH     PT_ATTRIBLENGTH / 4     /* max. Laenge eines Attributes */

/* class sizetable */

  #define ST_MAXENTRIES PT_MAXENTRIES   /* max. Eintraege in size_table */

#endif

/* class allocsize */

#ifdef ALLOCSIZE
  #define ALLOCSIZE_FILE        "/home/urz/fs18/dqs-3.1.1/partd/gc_allocated"
  #define SLEEP_ON_RETRY        1
  #define MAX_RETRY             5
#endif

/* class complex */

#ifdef COMPLEX
  #define GC_CONF_FILE          "/tmp/gc_config"
```

```
  #define ALLOC_FILE            "/tmp/gc_alloc"
  #define P_COMP_FILE           "/tmp/p_complex"
  #define A_COMP_FILE           "/tmp/a_complex"
  #define ALL_COMP_FILE         "/tmp/all_complex"
  #define COMMAND_P_COMP        "qconf3 -mc gc_p /tmp/p_complex"
  #define COMMAND_A_COMP        "qconf3 -mc gc_a /tmp/a_complex"
  #define COMMAND_ALL_COMP      "qconf3 -mc gc_all /tmp/all_complex"
  #define COMMAND_READ_CONFIG   "/home/user/MPCParix/bin/nrm -pc >/tmp/gc_config"
  #define COMMAND_READ_ALLOC    "/home/user/MPCParix/bin/nrm -pa >/tmp/gc_alloc"
#endif

/* class config */

#ifdef CONFIG
  #define CONFIG_FILE           "/home/urz/fs18/dqs-3.1.1/partd/conf_file"
  #define RESSOURCE_LENGTH      64
  #define RESS_PARTITION        "PARTITION"
  #define RESS_JOBTIME          "JOBTIME"
  #define DEFAULT_JOBTIME       5
  #define BUSINESS_TIME_START   6
  #define BUSINESS_TIME_STOP    19
  #define GC_SIZE               128
  #define MAX_ALLOC_B_TIME      64
#endif
```

A.2 Klasse array - array.h

Die Klasse *array* stellt grundlegende Funktionen zur Verwaltung von Feldern bereit. Die Elemente können anhand eines Schlüssels in das Feld einsortiert oder gefunden werden.
Die beiden vrtuellen Methoden *compare_key(...)* und *move_up_element(...)* müssen von erbenden Klassen überschrieben werden, da der Klasse *array* keine Informationen über die Struktur der Feldelemente besitzt.

```
#define  ARRAY
#include "defs.h"

class array
{
  private:
    int         entry_max;      /* maximale Feldgroesse */
    int         entry_count;    /* Anzahl der Eintraege */

  public:
                        array();
                        /* Konstruktor */
                void    set_maxentries(int n);
                        /* setzt <entry_max> auf <n> */
                int     count();
                        /* liefert <entry_count> */
                void    set_count(int n);
                        /* setzt <entry_count> auf <n> */
                int     find(void *key, int *r_code);
                        /* sucht nach <key>, liefert:
                           gefundene Position und setzt <r_code> = FOUND bzw.
                           Pos. an der <key> eingefuegt werden kann und setzt
                           <r_code> = NOT_FOUND */
                int     _insert(void *key);
                        /* fuegt leeres Element anhand von <key> ein
                           liefert dessen Position */

    /* die nachfolgenden Methoden muessen ueberschrieben werden ! */

    virtual     int     compare_key(void *key, int pos);
                        /* vergleicht <key> mit Schluessel des Elements an
                           Position <pos> und liefert Rueckgabewert
                           analog strcmp */
    virtual     void    move_up_element(int pos);
                        /* schiebt Element an Position <pos> nach <pos+1> */
};
```

A.3 Klassen parttable, attribtable, sizetable - tables.h

Die Klasse *parttable* dient der Speicherung der Partitionenangaben (siehe *partentry_type*) und wird zur Ermittlung von Partitionsattributen oder -größe einer physischen Partionsbezeichnung verwendet.
Die Klassen *attribtable* und *sizetable* werden zur für die Erstellung des Komplexes *gc_a*, der die Attribute und Größen freier Partitionen beinhaltet, benötigt.

```
#define  TABLES
#include "array.h"      /* Basisklasse */


struct partentry
{
  char  name[PT_NAMELENGTH];           /* phys. Partitionsbezeichnung */
  int   size;                          /* Partitionsgroesse (aus "p..") */
  char  attributes[PT_ATTRIBLENGTH];   /* Partitionsattribute */
};
typedef partentry       partentry_type;


class parttable:public array    /* Verwaltung der phys Partitionsbezeichnungen */
{
  private:
    partentry_type      part_table[PT_MAXENTRIES];

  public:
                                 parttable();
                partentry_type  *get(int pos);
                                 /* liefert Zeiger auf <pos> zurueck */
                int              insert(char *name, int size, char *attributes);
                                 /* fuegt <partentry> ein */

                                 /* von "array" ueberschrieben */
    virtual     void             move_up_element(int pos);
    virtual     int              compare_key(void *key, int pos);
};


typedef char    attrib_type[AT_LENGTH];


class attribtable: public array        /* Verwaltung der Attribute */
{
  private:
    attrib_type         attrib_table[AT_MAXENTRIES];
```

```
  public:
                                attribtable();
              attrib_type      *get(int pos);
              int               insert(char *attribute);
    virtual   void              move_up_element(int pos);
    virtual   int               compare_key(void *key, int pos);
};


class sizetable: public array           /* Verwaltung der Partitionengroessen */
{
  private:
    int                 size_table[ST_MAXENTRIES];

  public:
                        sizetable();
              int       get(int pos);
              int       insert(int size);
    virtual   void      move_up_element(int pos);
    virtual   int       compare_key(void *key, int pos);
};
```

A.4 Klasse allocsize - allocsize.h

Diese Klasse realisiert die Kommunikation des Dämons mit den Jobs über die gemeinsam genutzte Datei **ALLOCSIZE_FILE**. Von Seiten des Jobs wird bei Allokierung und Freigabe von Partitionen diese Datei mittels **inform_partd** ausgewertet.

```
#define         ALLOCSIZE
#include        "defs.h"


struct  as_entry        /* Struktur des "ALLOCSIZE_FILE" */
{
        int     allocated_size;         /* von DQS belegte Groesse */
        int     max_alloc_size;         /* max. zu belegende Groesse */
        int     time_to_wait;           /* Wartezeit bei fehlgeschlagener Allokierung
};
typedef as_entry as_entry_type;


class allocsize
{
  public:
    int set(as_entry_type *as_ptr);     /* legt ALLOCSIZE_FILE entsprechend
                                           <*as_ptr> an */
    int get();                          /* liest <allocated_size>
                                           aus ALLOCSIZE_FILE */
    int get(as_entry_type *as_ptr);     /* liest <*as_ptr>
                                           aus ALLOCSIZE_FILE */
    int modify_allocated_size(int size);        /* addiert <size> zu <allocated_size>*
    int set_max_alloc_size(int size);           /* setzt <max_alloc_size> auf <size> *
    int set_time_to_wait(int seconds);          /* setzt <time_to_wait> auf <size> */
};
```

A.5 Klasse complex - complex.h

Hier werden das Auswerten der GC-Konfiguration und der momentanen GC-Belegung sowie das Schreiben der verschiedenen Komplexe durchgeführt.

```
#include "allocsize.h"
#define  COMPLEX
#include "tables.h"


class complex
{
  protected:
    sizetable            st;    /* Die Tabellen werden fuer die */
    parttable            pt;    /* Zuordnung zu den verschiedenen */
    attribtable          at;    /* "Komplexen" beoetigt */
    allocsize            as;
    partentry_type      *pt_ptr[PT_MAXENTRIES];
    int                  freepart_count;

  public:
    int read_gc_config();       /* liest GC-Konfiguration ein */
    int read_alloc(int max_size);       /* sucht nach freien Partitionen */
    int set_p(int max_size, int bt);    /* schreibt "Komplex" fuer phys. Partititio
    int set_a(int max_size, int bt);    /* schreibt "Komplex" fuer Attributbezeichn
    int set_all(int max_size);  /* schreibt "Komplex" fuer Partitionen mit Groesse
};
```

A.6 Klasse config - config.h

Die Klasse *config* liest die Dämon-Konfiguration aus **CONFIG_FILE** ein.

```
#define  CONFIG
#include "defs.h"

struct ss_timer
{
  int   start;          /* Startzeit */
  int   stop;           /* Stopzeit */
};
typedef ss_timer        ss_time_type;


class config
{
  protected:
    ss_time_type        day_timer[7];           /* Geschaeftszeiten aller Tage */
    char                ress_partition[RESSOURCE_LENGTH];
    char                ress_jobtime[RESSOURCE_LENGTH];
    int                 gc_size;                /* GC Groesse */
    int                 max_alloc_b_time;       /* max. Belegungsgr. zur Geschaeftszei

  public:
    config();   /* liest 'CONFIG_FILE' und setzt Variablen */
};
```

A.7 Klasse daemon - daemon.h

Diese Klasse realisiert die zeitgesteuerte Prioritätensteuertung, Job-Laufzeitüberwachung und Komplexmodifikation.

```
extern "C"
{
  #include <h.h>                          /* Include-Dateien von DQS */
  #include <def.h>
  #include <dqs.h>
  #include <struct.h>
  #include <func.h>
  #include <globals.h>
  #include <dqs_errno.h>
}

#include <time.h>

#include "complex.h"
#define  DAEMON
#include "config.h"


class daemon:public complex, public config
{
  private:
    int         sleep_time;               /* Wartezeit zwischen zwei Durchlaeufen */
    time_t      time_actual;              /* aktuelle Zeit */
    time_t      time_modify_gc;           /* Zeit der naechsten Geschaeftszeitumschal
    time_t      time_sync;                /* Zeit der naechsten Synchronisation */
    int         business_time;            /* log. Variable */
    int         act_alloc;                /* GC-Gesamtgroesse laufender DQS-Job's */

    void        modify_times();
                /* berechnet 'time_modify_gc' und modifiziert 'business_time' */
    void        check_running_job(dqs_job_type *job);
                /* testet Job auf Laufzeitueberschreitung */
    void        modify_priority(dqs_job_type *job);
                /* modifiziert die Prioritaet wartender Jobs */
    int         new_priority(u_long32 w_time, int size, u_long32 time);
                /* berechnet die eigentliche Prioritaet */
    int         check_gc_job(dqs_job_type *job, int *size, int *jobtime);
                /* testet, ob Job ein GC-Job ist */

  public:
    int         init();
```

```
    int         make_daemon();
    void        syncronize(int allocated_size);
    void        scan_jobs();
    void        run();
};
```

B Warteschlangendefinitionen

B.1 Queue gc_physic

```
Q_name           gc_physic
hostname         abel.hrz.tu-chemnitz.de
seq_no           100
load_masg        1
load_alarm       175
priority         0
type             batch
rerun            FALSE
quantity         8
tmpdir           /tmp
shell            /bin/bash
klog             /usr/afsws/bin/klog
reauth_time      6000
notify           60
owner_list       NONE
user_acl         NONE
xuser_acl        NONE
subordinate_list NONE
complex_list     gc_p
s_rt             7fffffff
h_rt             7fffffff
s_cpu            7fffffff
h_cpu            7fffffff
s_fsize          7fffffff
h_fsize          7fffffff
s_data           7fffffff
h_data           7fffffff
s_stack          7fffffff
h_stack          7fffffff
s_core           7fffffff
h_core           7fffffff
s_rss            7fffffff
h_rss            7fffffff
```

B.2 Queue gc_attribute

```
Q_name           gc_attribute
hostname         abel.hrz.tu-chemnitz.de
seq_no           101
load_masg        1
load_alarm       175
priority         0
type             batch
rerun            FALSE
quantity         8
tmpdir           /tmp
shell            /bin/bash
klog             /usr/afsws/bin/klog
reauth_time      6000
notify           60
owner_list       NONE
user_acl         NONE
xuser_acl        NONE
subordinate_list NONE
complex_list     gc_a
s_rt             7fffffff
h_rt             7fffffff
s_cpu            7fffffff
h_cpu            7fffffff
s_fsize          7fffffff
h_fsize          7fffffff
s_data           7fffffff
h_data           7fffffff
s_stack          7fffffff
h_stack          7fffffff
s_core           7fffffff
h_core           7fffffff
s_rss            7fffffff
h_rss            7fffffff
```

B.3 Queue gc_128

```
Q_name           gc_128
hostname         abel.hrz.tu-chemnitz.de
seq_no           102
load_masg        1
load_alarm       175
priority         0
type             batch
rerun            FALSE
quantity         1
tmpdir           /tmp
shell            /bin/bash
klog             /usr/afsws/bin/klog
reauth_time      6000
notify           60
owner_list       NONE
user_acl         NONE
xuser_acl        NONE
subordinate_list NONE
complex_list     gc_all
s_rt             7fffffff
h_rt             7fffffff
s_cpu            7fffffff
h_cpu            7fffffff
s_fsize          7fffffff
h_fsize          7fffffff
s_data           7fffffff
h_data           7fffffff
s_stack          7fffffff
h_stack          7fffffff
s_core           7fffffff
h_core           7fffffff
s_rss            7fffffff
h_rss            7fffffff
```

Literatur

[1] PARIX User's Guide http::/www.tu-chemnitz.de/home/jwa/para/usersguide.ps

[2] PARIX Release Notes http::/www.tu-chemnitz.de/home/jwa/para/releasenotes.ps

[3] Kurze Einführung in die Nutzung des Parallelrechners http::/www.tu-chemnitz.de/home/jwa/parause.html

[4] Florida State University http://www.fsu.edu/~pasko/dqs.html

[5] DQS User Interface http::/www.tu-chemnitz.de/global-text/doc/dqs-3.1.1/DQS_user_interface.ps

[6] DQS 3.1.1 Readme/Installation Manual http::/www.tu-chemnitz.de/global-text/doc/dqs-3.1.1/DQS_readme.ps

[7] CCS Info-Tool http::/www/uni-paderborn.de/pcpc/work/ccs/ccs.html

[8] B. Bauer, F. Ramme: "A general purpose Description Language", Reihe Informatik aktuell, Hrsg: R. Grebe, M. Springer-Verlag, (Berlin), 1991, pp., 68-75.

[9] F. Ramme, T. Roemke "Ressource Description Language - Language Definition", Paderborn Center for Parallel Computing, int. Report, 1992, verfügbar als Postscript-File unter ~/pccs/public/ccs/doc/rdldef.ps